Naval Command,
Control and Ocean
Surveillance Center RDT&E Division San Diego, CA
92152-5001

Technical Document 2180
December 1990

Proposed Standard for a Generic Package of Elementary Functions for Ada

Draft 1.2

ISO-IEC/JTC1/SC22/WG9 (Ada)
Numerics Rapporteur Group

G. Myers, Chair

Technical Document 2180
December 1990

Proposed Standard for a Generic Package of Elementary Functions for Ada

Draft 1.2

ISO-IEC/JTC1/SC22/WG9 (Ada)
Numerics Rapporteur Group

G. Myers, Chair

NAVAL OCEAN SYSTEMS CENTER
San Diego, California 92152–5000

J. D. FONTANA, CAPT, USN
Commanding Officer

R. T. SHEARER
Acting Technical Director

ADMINISTRATIVE INFORMATION

Work for this report was performed by Advanced Concepts and Systems Technology Division, NCCOSC RDT&E Division, San Diego, California 92152–5001 during the period of 1988–1994. The work was funded by the Ada Joint Program Office.

Released by
G. Myers, Code 4104
Technical Staff

Under authority of
R. B. Volker, Head
Advanced Concepts and
Systems Technology Division

ACKNOWLEDGMENTS

The author acknowledges the contribution of the Ada Europe Numerics Working Group, SIGAda Numerics Working Group, and the ISO Numerics Rapporteur Group.

The technical report/document was compiled through the contribution of members of those groups. Ken Dritz of Argonne National Laboratory is the technical editor for this ISO standard.

LR

Generic Package of Elementary Functions (GPEF)

Description

The proposed standard for the Generic Package of Elementary Functions (GPEF) represents the work of a large number of people in both the United States and Europe who have collaborated to develop specifications for packages of Ada mathematical functions. This development has been difficult and lengthy. The exceptional dedication and perseverance of these people have resulted in the completed specifications for two packages—GPEF, and the Generic Package of Primitive Functions (GPPF) for Ada.

GPEF is the specification for certain elementary functions. They are square root, logarithm and exponential functions and the exponentiation operator; the trigonometric functions for sine, cosine, tangent and cotangent and their inverses; and the hyperbolic functions for sine, cosine, tangent, and cotangent together with their inverses.

Background

The Ada-Europe Numerics Working Group (A-ENWG) was formed in 1984 about the same time that an early study proposing standard mathematical packages in Ada was undertaken by Symm and Kok. In 1986, the Numerics Working Group (NUMWG) of the Association of Computing Machinery's Special Interest Group on Ada (ACM SIGAda) was formed, and has met every few months since. During the 1980s, members of A-ENWG met on a regular basis with the NUMWG so that close cooperation was achieved on developing specifications that were joint effort of both groups. The A-ENWG has not met for some three years, but the NUMWG continues informal liaison with key European Ada individuals on continuing work.

Current Status of Standardization

The proposed standards for GPEF and GPPF have been adopted by the Numerics Rapporteur Group (NRG), a subcommittee of Working Group 9 (Ada) of Subcommittee 22 of Joint Technical Committee 1 of the International Organization for Standardization-International Electrotechnical Commission (ISO-IEC JTC1/SC22/WG9 (Ada). WG9 has approved both proposed standards and has forwarded them to SC22 for voting. GPEF has been accepted as Draft International Standard (DIS) 11430 and GPPF has been approved as DIS 11729. The completion of editorial formatting of both documents for final publication as international standards is expected this year.

Gilbert Myers
Chair, ACM SIGAda NUMWG, ISO NRG
May 10, 1994

This document defines the specification of a generic package of elementary functions called `GENERIC_ELEMENTARY_FUNCTIONS` and the specification of a package of related exceptions called `ELEMENTARY_FUNCTIONS_EXCEPTIONS`. It does not define the body of the former. No body is required for the latter.

Deriving its content from a joint proposal of the ACM SIGAda Numerics Working Group and the Ada-Europe Numerics Working Group, the WG9 Numerics Rapporteur Group submitted Draft 1.0 of this proposed standard to WG9 in March, 1989. WG9 approved Draft 1.0 in June, 1989, subject to minor revisions and improvements; these changes resulted in Draft 1.1. Initial processing by SC22 and further work by the NRG resulted in additional improvements, leading to this current version, Draft 1.2, intended for further processing as an international standard.

The generic package described here is intended to provide the basic mathematical routines from which portable, reusable applications can be built. The standard serves a broad class of applications with reasonable ease of use, while demanding implementations that are of high quality, capable of validation, and also practical given the state of the art.

The two specifications included in this document are presented as compilable Ada specifications followed by explanatory text in numbered sections. The explanatory text is an integral part of the standard, with the exception of the following items:

(1) in Section 13, examples of common usage of the elementary functions (under the heading *Usage* associated with each function);

(2) also in Section 13, notes (under the heading *Notes* associated with some of the functions); and

(3) notes (labeled as such) presented at the end of any numbered section.

The word "may," as used in this document, consistently means "is allowed to" (or "are allowed to"). It is used only to express permission, as in the commonly occurring phrase "an implementation may"; other words (such as "can," "could," or "might") are used to express ability, possibility, capacity, or consequentiality.

This proposal was prepared under the leadership of G. Myers with contributions by the following individuals, listed in alphabetical order: A. Adamson, J. G. P. Barnes, W. J. Cody, P. M. Cohen, S. G. Cohen, K. W. Dritz, B. Ford, J. B. Goodenough, G. S. Hodgson, J. Kok, R. F. Mathis, T. G. Mattson, B. T. Smith, J. S. Squire, P. T. P. Tang, W. A. Whitaker, D. T. Winter, and M. Woodger. Many others contributed through international meetings and electronic mail reviews. Organizations lending support to this effort were the Naval Ocean Systems Center, Argonne National Laboratory, Westinghouse Electric Corporation, Numerical Algorithms Group, Centrum voor Wiskunde en Informatica, Software Productivity Consortium, Contel, Martin Marietta, the Software Engineering Institute of Carnegie Mellon University, Quantitative Technology Corporation, IBM, and Alsys.

Bibliography

[1] M. Abramowitz and I. Stegun. *Handbook of Mathematical Functions*. U. S. Government Printing Office, Washington, D. C., 1964.

[2] W. J. Cody and W. Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, 1980.

[3] K. W. Dritz. Rationale for the Proposed Standard for a Generic Package of Elementary Functions for Ada. ANL Report ANL-89/2 Rev. 1, Argonne National Laboratory, Argonne, Illinois, October 1989. A later (December 1990) revision is available from the author.

[4] B. Ford, J. Kok, and M. W. Rogers, editors. *Scientific Ada*. Cambridge University Press, Cambridge, 1986.

[5] F. B. Hildebrand. *Introduction to Numerical Analysis*. McGraw-Hill, 1956.

[6] IEEE. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std. 754-1985, IEEE, New York, 1985.

[7] IEEE. IEEE Standard for Radix-Independent Floating-Point Arithmetic. ANSI/IEEE Std. 854-1987, IEEE, New York, 1987.

[8] J. Kok. Proposal for Standard Mathematical Packages in Ada. CWI Report NM-R8718, Centrum voor Wiskunde en Informatica, Amsterdam, November 1987.

[9] R. F. Mathis. Elementary Functions Packages for Ada. In *Proc. 1987 ACM SIGAda International Conference on the Ada Programming Language* (special issue of *Ada Letters*), pages 95–100, December 1987.

[10] U. S. Department of Defense, Ada Joint Program Office. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A. U. S. Government Printing Office, Washington, D. C., 1983. Also adopted by ISO as ISO/8652-1987, Programming Languages—Ada.

```
package ELEMENTARY_FUNCTIONS_EXCEPTIONS is
   ARGUMENT_ERROR : exception;
end ELEMENTARY_FUNCTIONS_EXCEPTIONS;

with ELEMENTARY_FUNCTIONS_EXCEPTIONS;
generic
   type FLOAT_TYPE is digits <>;
package GENERIC_ELEMENTARY_FUNCTIONS is

   function SQRT    (X         : FLOAT_TYPE)        return FLOAT_TYPE;
   function LOG     (X         : FLOAT_TYPE)        return FLOAT_TYPE;
   function LOG     (X, BASE   : FLOAT_TYPE)        return FLOAT_TYPE;
   function EXP     (X         : FLOAT_TYPE)        return FLOAT_TYPE;
   function "**"    (X, Y      : FLOAT_TYPE)        return FLOAT_TYPE;

   function SIN     (X         : FLOAT_TYPE)        return FLOAT_TYPE;
   function SIN     (X, CYCLE  : FLOAT_TYPE)        return FLOAT_TYPE;
   function COS     (X         : FLOAT_TYPE)        return FLOAT_TYPE;
   function COS     (X, CYCLE  : FLOAT_TYPE)        return FLOAT_TYPE;
   function TAN     (X         : FLOAT_TYPE)        return FLOAT_TYPE;
   function TAN     (X, CYCLE  : FLOAT_TYPE)        return FLOAT_TYPE;
   function COT     (X         : FLOAT_TYPE)        return FLOAT_TYPE;
   function COT     (X, CYCLE  : FLOAT_TYPE)        return FLOAT_TYPE;

   function ARCSIN  (X         : FLOAT_TYPE)        return FLOAT_TYPE;
   function ARCSIN  (X, CYCLE  : FLOAT_TYPE)        return FLOAT_TYPE;
   function ARCCOS  (X         : FLOAT_TYPE)        return FLOAT_TYPE;
   function ARCCOS  (X, CYCLE  : FLOAT_TYPE)        return FLOAT_TYPE;
   function ARCTAN  (Y         : FLOAT_TYPE;
                     X         : FLOAT_TYPE := 1.0) return FLOAT_TYPE;
   function ARCTAN  (Y         : FLOAT_TYPE;
                     X         : FLOAT_TYPE := 1.0;
                     CYCLE     : FLOAT_TYPE)        return FLOAT_TYPE;
   function ARCCOT  (X         : FLOAT_TYPE;
                     Y         : FLOAT_TYPE := 1.0) return FLOAT_TYPE;
   function ARCCOT  (X         : FLOAT_TYPE;
                     Y         : FLOAT_TYPE := 1.0;
                     CYCLE     : FLOAT_TYPE)        return FLOAT_TYPE;

   function SINH    (X         : FLOAT_TYPE)        return FLOAT_TYPE;
   function COSH    (X         : FLOAT_TYPE)        return FLOAT_TYPE;
   function TANH    (X         : FLOAT_TYPE)        return FLOAT_TYPE;
   function COTH    (X         : FLOAT_TYPE)        return FLOAT_TYPE;

   function ARCSINH (X         : FLOAT_TYPE)        return FLOAT_TYPE;
   function ARCCOSH (X         : FLOAT_TYPE)        return FLOAT_TYPE;
   function ARCTANH (X         : FLOAT_TYPE)        return FLOAT_TYPE;
   function ARCCOTH (X         : FLOAT_TYPE)        return FLOAT_TYPE;

   ARGUMENT_ERROR : exception renames ELEMENTARY_FUNCTIONS_EXCEPTIONS.ARGUMENT_ERROR;

end GENERIC_ELEMENTARY_FUNCTIONS;
```

CONTENTS

1. Purpose

This standard provides certain elementary mathematical functions. They were chosen because of their widespread utility in various application areas; moreover, they are needed to support general floating-point usage and to support generic packages for complex arithmetic and complex functions.

2. Functions provided

The following twenty mathematical functions are provided:

```
SQRT     LOG      EXP      "**"
SIN      COS      TAN      COT
ARCSIN   ARCCOS   ARCTAN   ARCCOT
SINH     COSH     TANH     COTH
ARCSINH  ARCCOSH  ARCTANH  ARCCOTH
```

These are the square root (`SQRT`), logarithm (`LOG`), and exponential (`EXP`) functions and the exponentiation operator (`**`); the trigonometric functions for sine (`SIN`), cosine (`COS`), tangent (`TAN`), and cotangent (`COT`) and their inverses (`ARCSIN`, `ARCCOS`, `ARCTAN`, and `ARCCOT`); and the hyperbolic functions for sine (`SINH`), cosine (`COSH`), tangent (`TANH`), and cotangent (`COTH`) together with their inverses (`ARCSINH`, `ARCCOSH`, `ARCTANH`, and `ARCCOTH`).

3. Instantiations

This standard describes a generic package, `GENERIC_ELEMENTARY_FUNCTIONS`, which the user must instantiate to obtain a package. It has one generic formal parameter, which is a generic formal type named `FLOAT_TYPE`. At instantiation, the user must specify a floating-point subtype as the generic actual parameter to be associated with `FLOAT_TYPE`; it is referred to below as the "generic actual type." This type is used as the parameter and result type of the functions contained in the generic package.

Depending on the implementation, the user may or may not be allowed to specify a generic actual type having a range constraint (cf. Section 4). If allowed, such a range constraint will have the usual effect of causing `CONSTRAINT_ERROR` to be raised when an argument outside the user's range is passed in a call to one of the functions, or when one of the functions attempts to return a value outside the user's range. Allowing the generic actual type to have a range constraint also has some implications for implementors.

In addition to the body of the generic package itself, implementors may provide (non-generic) library packages that can be used just like instantiations of the generic package for the predefined floating-point types. The name of a package serving as a replacement for an instantiation of `GENERIC_ELEMENTARY_FUNCTIONS` for the predefined type `FLOAT` should be `ELEMENTARY_FUNCTIONS`; for `LONG_FLOAT` and `SHORT_FLOAT`, the names should be `LONG_ELEMENTARY_FUNCTIONS` and `SHORT_ELEMENTARY_FUNCTIONS`, respectively; etc. When such a package is used in an application in lieu of an instantiation of `GENERIC_ELEMENTARY_FUNCTIONS`, it must have the semantics implied by this standard for an instantiation of the generic package.

4. Implementations

Portable implementations of the body of `GENERIC_ELEMENTARY_FUNCTIONS` are strongly encouraged. However, implementations are not required to be portable. In particular, an implementation of the standard in Ada may use pragma `INTERFACE` or other pragmas, unchecked conversion, machine-code insertions, or other machine-dependent techniques as desired.

An implementation is allowed to limit the precision it supports (by stating an assumed maximum value for SYSTEM'MAX_DIGITS), since portable implementations would not, in general, be possible otherwise. An implementation is also allowed to make other reasonable assumptions about the environment in which it is to be used, but only when necessary in order to match algorithms to hardware characteristics in an economical manner. All such limits and assumptions must be clearly documented. By convention, an implementation of GENERIC_ELEMENTARY_FUNCTIONS is said not to conform to this standard in any environment in which its limits or assumptions are not satisfied, and this standard does not define its behavior in that environment. In effect, this convention delimits the portability of implementations.

An implementation is allowed to impose a restriction that the generic actual type must not have a range constraint that reduces the range of allowable values. If it does impose this restriction, then the restriction must be documented, and the effects of violating the restriction must be one of the following:

(1) Compilation of a unit containing an instantiation of GENERIC_ELEMENTARY_FUNCTIONS is rejected.

(2) CONSTRAINT_ERROR or PROGRAM_ERROR is raised during the elaboration of an instantiation of GENERIC_ELEMENTARY_FUNCTIONS.

Conversely, if an implementation does not impose the restriction, then it must not allow such a range constraint, when included with the user's actual type, to interfere with the internal computations of the functions; that is, if the argument and result are within the range of the type, then the implementation must return the result and must not raise an exception (such as CONSTRAINT_ERROR).

An implementation must function properly in a tasking environment. Apart from the obvious restriction that an implementation of GENERIC_ELEMENTARY_FUNCTIONS must avoid declaring variables that are global to the functions, no special constraints are imposed on implementations. Nothing in this standard requires the use of such global variables.

Some hardware and their accompanying Ada implementations have the capability of representing and discriminating between positively and negatively signed zeros as a means, e.g., of preserving the sign of an infinitesimal quantity that has underflowed to zero. This standard allows implementations of GENERIC_ELEMENTARY_FUNCTIONS to exploit that capability, when available, so as to exhibit continuity in the results of ARCTAN and ARCCOT as certain limits are approached. At the same time, it accommodates implementations in which that capability is unavailable. Because a definition of what comprises the capability of representing and distinguishing signed zeros is beyond the scope of this standard, implementations are allowed the freedom not to exploit the capability, even when it is available. This standard accommodates the various choices allowed to implementations by, e.g., defining the results of ARCTAN and ARCCOT to be an implementation dependent choice between two values in certain limited cases involving a zero argument. An implementation must exercise its choice consistently, either exploiting signed-zero behavior everywhere or nowhere in this package. In addition, an implementation must document its behavior with respect to signed zeros.

5. Exceptions

One exception, ARGUMENT_ERROR, is declared in GENERIC_ELEMENTARY_FUNCTIONS. This exception is raised by a function in the generic package only when the argument(s) of the function violate one or more of the conditions given in the function's domain definition (cf. Section 8). Note that these conditions are related only to the mathematical definition of the function and are therefore implementation independent.

The ARGUMENT_ERROR exception is declared as a renaming of the exception of the same name declared in the ELEMENTARY_FUNCTIONS_EXCEPTIONS package. Thus, this exception distinguishes neither between different kinds of argument errors, nor between different functions, nor between different instantiations of GENERIC_ELEMENTARY_FUNCTIONS.

Besides ARGUMENT_ERROR, the only exceptions allowed during a call to a function in GENERIC_ELEMENTARY_FUNCTIONS are predefined exceptions, as follows:

(1) Virtually any predefined exception is possible during the evaluation of an argument of a function in GENERIC_ELEMENTARY_FUNCTIONS. For example, NUMERIC_ERROR, CONSTRAINT_ERROR, or even PROGRAM_ERROR could be raised if an argument has an undefined value; and, as stated in Section 3, if the implementation allows range constraints in the generic actual type, then CONSTRAINT_ERROR will be raised when the value of an argument lies outside the range of the user's generic actual type. Additionally, STORAGE_ERROR could be raised, e.g. if insufficient storage is available to perform the call. All these exceptions are raised before the body of the function is entered and therefore have no bearing on implementations of GENERIC_ELEMENTARY_FUNCTIONS.

(2) Also as stated in Section 3, if the implementation allows range constraints in the generic actual type, then CONSTRAINT_ERROR will be raised when a function in GENERIC_ELEMENTARY_FUNCTIONS attempts to return a value outside the range of the user's generic actual type. The exception raised for this reason must be propagated to the caller of the function.

(3) Whenever the arguments of a function are such that a result permitted by the accuracy requirements would exceed FLOAT_TYPE'SAFE_LARGE in absolute value, as formalized below in Section 11, an implementation may raise (and must then propagate to the caller) the exception specified by Ada for signaling overflow.

(4) Whenever the arguments of a function are such that the corresponding mathematical function is infinite (cf. Section 11), an implementation must raise and propagate to the caller the exception specified by Ada for signaling division by zero.

(5) Once execution of the body of a function has begun, an implementation may propagate STORAGE_ERROR to the caller of the function, but only to signal the exhaustion of storage. Similarly, once execution of the body of a function has begun, an implementation may propagate PROGRAM_ERROR to the caller of the function, but only to signal errors made by the user of GENERIC_ELEMENTARY_FUNCTIONS.

No exception is allowed during a call to a function in GENERIC_ELEMENTARY_FUNCTIONS except those permitted by the foregoing rules. In particular, for arguments for which all results satisfying the accuracy requirements remain less than or equal to FLOAT_TYPE'SAFE_LARGE in absolute value, a function must locally handle an overflow occurring during the computation of an intermediate result, if such an overflow is possible, and not propagate an exception signaling that overflow to the caller of the function.

The only exceptions allowed during an instantiation of GENERIC_ELEMENTARY_FUNCTIONS, including the execution of the optional sequence of statements in the body of the instance, are CONSTRAINT_ERROR, PROGRAM_ERROR, and STORAGE_ERROR, and then only for the reasons given below. The raising of CONSTRAINT_ERROR during instantiation is only allowed when the implementation imposes the restriction that the generic actual type must not have a range constraint, and the user violates that restriction (it may, in fact, be an inescapable consequence of the violation). The raising of PROGRAM_ERROR during instantiation is only allowed for the purpose of signaling errors made by the user—for example, violation of this same restriction, or of other limitations of the implementation. The raising of STORAGE_ERROR during instantiation is only allowed for the purpose of signaling the exhaustion of storage.

Note:

In the Ada Reference Manual, the exception specified for signaling overflow or division by zero is NUMERIC_ERROR, but AI-00387 replaces that by CONSTRAINT_ERROR.

6. Arguments outside the range of safe numbers

The current Ada standard fails to define the result safe interval of any basic or predefined operation of a real subtype when the absolute value of one of its operands exceeds the largest safe number of the operand subtype. (The failure to define a result in this case occurs because no safe interval is defined for the operand in

question.) In order to avoid imposing requirements that would, consequently, be more stringent than those of Ada itself, this standard likewise does not define the result of a contained function when the absolute value of one of its arguments exceeds FLOAT_TYPE'SAFE_LARGE. All of the accuracy requirements and other provisions of the following sections are understood to be implicitly qualified by the assumption that function arguments are less than or equal to FLOAT_TYPE'SAFE_LARGE in absolute value.

7. Method of specification of functions

Some of the functions have two overloaded forms. For each form of a function covered by this standard, the function is specified by its parameter and result type profile, the domain of its argument(s), its range, and the accuracy required of its implementation. The meaning of, and conventions applicable to, the domain, range, and accuracy specifications are described below.

8. Domain definitions

The specification of each function covered by this standard includes, under the heading *Domain*, a characterization of the argument values for which the function is mathematically defined. It is expressed by inequalities or other conditions which the arguments must satisfy to be valid. The phrase "mathematically unbounded" in a domain definition indicates that all representable values of the argument are valid. Whenever the arguments fail to satisfy all the conditions, the implementation must raise ARGUMENT_ERROR. It must not raise that exception if all the conditions are satisfied.

Inability to deliver a result for valid arguments because the result overflows, for example, must not raise ARGUMENT_ERROR, but must be treated in the same way that Ada defines for its predefined floating-point operations (cf. Section 11); after all, one of these operations causes the overflow.

Note:

Unbounded portions of the domains of the functions EXP, "**", SINH, and COSH, which are "expansion" functions with unbounded or semi-unbounded mathematical domains, are unexploitable because the corresponding function values (satisfying the accuracy requirements) cannot be represented. Their "usable domains," i.e. the portions of the mathematical domains given in their domain definitions that are exploitable in the sense that they produce representable results, are given by the notes accompanying their specifications. Because of permitted variations in implementations, these usable domains can only be stated approximately. In a similar manner, functions such as TAN and COT with periodic "poles" in their domains might or might not (depending on the implementation) have small unusable portions of their domains in the vicinities of the poles. Also, range constraints in the user's generic actual type can, by narrowing a function's range, make further portions of the function's domain unusable.

9. Range definitions

The usual mathematical meaning of the "range" of a function is the set of values into which the function maps the values in its domain. Some of the functions covered by this standard (for example, ARCSIN) are mathematically multivalued, in the sense that a given argument value can be mapped by the function into many different result values. By means of range restrictions, this standard imposes a uniqueness requirement on the results of multivalued functions, thereby reducing them to single-valued functions.

Some of the functions covered by this standard (for example, EXP) have asymptotic behavior for extremely positive or negative arguments. Although there is no finite argument for which such a function can mathematically yield its asymptotic limit, that limit is always included in its range here, and it is an allowed result of the implemented function, in recognition of the fact that the limit value itself could be closer to the mathematical result than any other representable value.

The range of each function is shown under the heading *Range* in the specifications. Range definitions take the form of inequalities limiting the function value. An implementation must not exceed a limit of the range when that limit is a safe number of `FLOAT_TYPE` (like `0.0`, `1.0`, or `CYCLE/4.0` for certain values of `CYCLE`). On the other hand, when a range limit is not a safe number of `FLOAT_TYPE` (like π, or `CYCLE/4.0` for certain other values of `CYCLE`), an implementation is allowed to exceed the range limit, but it is not allowed to exceed the safe number of `FLOAT_TYPE` next beyond the range limit in the direction away from the interior of the range; this is in general the best that can be expected from a portable implementation. Effectively, therefore, range definitions have the added effect of imposing accuracy requirements on implementations above and beyond those presented under the heading *Accuracy* in the specifications (cf. Section 10).

The phrase "mathematically unbounded" in a range definition indicates that the range of values of the function is not bounded by its mathematical definition. It also implies that the function is not mathematically multivalued.

Note:

Unbounded portions of the ranges of the functions `SQRT`, `LOG`, `ARCSINH`, and `ARCCOSH`, which are "contraction" functions with unbounded or semi-unbounded mathematical ranges, are unreachable because the corresponding arguments cannot be represented. Their "reachable ranges," i.e. the portions of the mathematical ranges given in their range definitions that are reachable through appropriate arguments, are given by the notes accompanying their specifications. Because of permitted variations in implementations, these reachable ranges can only be stated approximately. Also, range constraints in the user's generic actual type can, by narrowing a function's domain, make further portions of the function's range unreachable.

10. Accuracy requirements

Because they are implemented on digital computers with only finite precision, the functions provided in this generic package can, at best, only approximate the corresponding mathematically defined functions.

The accuracy requirements contained in this standard define the latitude that implementations are allowed in approximating the intended precise mathematical result with floating-point computations. Accuracy requirements of two kinds are stated under the heading *Accuracy* in the specifications. Additionally, range definitions stated under the heading *Range* impose requirements that constrain the values implementations may yield, so the range definitions are another source of accuracy requirements (the precise meaning of a range limit that is not a safe number of `FLOAT_TYPE`, as an accuracy requirement, is discussed above in Section 9). Every result yielded by a function is subject to all of the function's applicable accuracy requirements, except in the one case described in Section 12, below. In that case, the result will satisfy a small absolute error requirement in lieu of the other accuracy requirements defined for the function.

The first kind of accuracy requirement used under the heading *Accuracy* in the specifications is a bound on the relative error in the computed value of the function, which must hold (except as provided by the rules in Sections 11 and 12) for all arguments satisfying the conditions in the domain definition, providing the mathematical result is nonzero. For a given function f, the relative error $re(X)$ in a computed result $F(X)$ at the argument X is defined in the usual way,

$$re(X) = \left| \frac{F(X) - f(X)}{f(X)} \right|$$

providing the mathematical result $f(X)$ is finite and nonzero. (The relative error is not defined when the mathematical result is infinite or zero.) For each function, the bound on the relative error is identified under the heading *Accuracy* as its maximum relative error.

The second kind of accuracy requirement used under the heading *Accuracy* in the specifications is a stipulation, in the form of an equality, that the implementation must deliver "prescribed results" for certain special arguments. It is used for two purposes: to define the computed result to be zero when the relative error is undefined, i.e.,

when the mathematical result is zero, and to strengthen the accuracy requirements at special argument values. When such a prescribed result is a safe number of FLOAT_TYPE (like 0.0, 1.0, or CYCLE/4.0 for certain values of CYCLE), an implementation must deliver that result. On the other hand, when a prescribed result is not a safe number of FLOAT_TYPE (like π, or CYCLE/4.0 for certain other values of CYCLE), an implementation may deliver any value in the surrounding safe interval. Prescribed results take precedence over maximum relative error requirements but never contravene them.

Range definitions, under the heading *Range* in the specifications, are an additional source of accuracy requirements, as stated above in Section 9. As an accuracy requirement, a range definition (other than "mathematically unbounded") has the effect of eliminating some of the values permitted by the maximum relative error requirements, e.g., those outside the range.

11. Overflow

Floating-point hardware is typically incapable of representing numbers whose absolute value exceeds some implementation-defined maximum. For the type FLOAT_TYPE, that maximum will be at least FLOAT_TYPE'SAFE_LARGE. For the functions defined by this standard, whenever the maximum relative error requirements permit a result whose absolute value is greater than FLOAT_TYPE'SAFE_LARGE, the implementation may

(1) yield any result permitted by the maximum relative error requirements, or

(2) raise the exception specified by Ada for signaling overflow.

An implementation must raise the exception specified by Ada for signaling division by zero in the following specific cases:

(1) LOG(X) when X = 0.0;

(2) LOG(X, BASE) when X = 0.0;

(3) X ** Y when X = 0.0 and Y < 0.0;

(4) TAN(X, CYCLE) when X = $(2k+1) \cdot$ CYCLE/4.0, for integer k;

(5) COT(X) when X = 0.0;

(6) COT(X, CYCLE) when X = $k \cdot$ CYCLE/2.0, for integer k;

(7) COTH(X) when X = 0.0;

(8) ARCTANH(X) when X = $\pm$1.0; and

(9) ARCCOTH(X) when X = $\pm$1.0.

(At these arguments, the corresponding mathematical functions are infinite.)

Notes:

This rule permits an implementation to raise an exception, instead of delivering a result, for arguments for which the mathematical result is close to but does not exceed FLOAT_TYPE'SAFE_LARGE in absolute value. Such arguments must necessarily be very close to an argument for which the mathematical result does exceed FLOAT_TYPE'SAFE_LARGE in absolute value. In general, this is the best that can be expected from a portable implementation with a reasonable amount of effort.

The rule is motivated by the behavior prescribed by the Ada Reference Manual for the predefined operations. That is, when the set of possible results of a predefined operation includes a number whose absolute value exceeds the implementation-defined maximum, the implementation is allowed to raise the exception specified for signaling overflow instead of delivering a result.

In the Ada Reference Manual, the exception specified for signaling overflow is NUMERIC_ERROR, but AI-00387 replaces that by CONSTRAINT_ERROR.

12. Underflow

Floating-point hardware is typically incapable of representing nonzero numbers whose absolute value is less than some implementation-defined minimum. For the type `FLOAT_TYPE`, that minimum will be at most `FLOAT_TYPE'SAFE_SMALL`. For the functions defined by this standard, whenever the maximum relative error requirements permit a result whose absolute value is less than `FLOAT_TYPE'SAFE_SMALL` and a prescribed result is not stipulated, the implementation may

(1) yield any result permitted by the maximum relative error requirements;

(2) yield any nonzero result having the correct sign and an absolute value less than or equal to `FLOAT_TYPE'SAFE_SMALL`; or

(3) yield zero.

Notes:

Whenever part (2) or (3) of this rule takes effect, the maximum relative error requirements are, in general, unachievable and are waived. In such cases, the computed result will exhibit an error which, while not necessarily small in relative terms, is small in absolute terms. The absolute error will, in these cases, be less than or equal to `FLOAT_TYPE'SAFE_SMALL`$/(1.0 - mre)$, where mre is the maximum relative error specified for the function under the heading *Accuracy*.

The rule permits an implementation to deliver a result violating the maximum relative error requirements for arguments for which the mathematical result equals or slightly exceeds `FLOAT_TYPE'SAFE_SMALL` in absolute value. Such arguments must necessarily be very close to an argument for which the mathematical result is less than `FLOAT_TYPE'SAFE_SMALL` in absolute value. In general, this is the best that can be expected from a portable implementation with a reasonable amount of effort.

The rule is motivated by the behavior prescribed by the Ada Reference Manual for predefined operations. That is, when the set of possible results of a predefined operation includes a nonzero number whose absolute value is less than the implementation-defined minimum, the implementation is allowed to yield zero or any nonzero number having the correct sign and an absolute value less than or equal to that minimum. An exception is never raised in this case.

13. Specifications of the functions

Under the heading *Definition* in each of the following subsections, the semantics of an Ada call to the function being defined is provided by a mathematical definition in the form of an approximation. The left-hand side (the function call) is set in the fixed-width font used throughout this document for program fragments. The right-hand side is to be interpreted as an exact mathematical formula; as such, it and similar mathematical formulas throughout this document employ standard mathematical symbols, notation, and fonts (except for variable names and some real literals, which are set in the fixed-width "program-fragment" font). The degree to which the function call on the left-hand side is allowed to approximate the value of the formula on the right-hand side is, of course, spelled out under the heading *Accuracy*, as discussed in Section 10.

13.1. SQRT — Square Root

Declaration:

```
function SQRT (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

SQRT(X) $\approx \sqrt{X}$

Usage:

```
Z := SQRT(X);
```

Domain:

X $\geq$ 0.0

Range:

SQRT(X) $\geq$ 0.0

Accuracy:

(a) Maximum relative error = 2.0 · FLOAT_TYPE'BASE'EPSILON
(b) SQRT(0.0) = 0.0

Notes:

The upper bound of the reachable range of SQRT is approximately given by

$$\text{SQRT(X)} \leq \sqrt{\text{FLOAT\_TYPE'SAFE\_LARGE}}$$

13.2. LOG — Natural Logarithm

Declaration:

```
function LOG (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

LOG(X) $\approx \log_e$ X

Usage:

```
Z := LOG(X);    -- natural logarithm
```

Domain:

$X \geq 0.0$

Range:

Mathematically unbounded

Accuracy:

(a) Maximum relative error = `4.0 · FLOAT_TYPE'BASE'EPSILON`

(b) `LOG(1.0) = 0.0`

Notes:

(a) The reachable range of `LOG` is approximately given by

$$\log_e \texttt{FLOAT\_TYPE'SAFE\_SMALL} \leq \texttt{LOG(X)} \leq \log_e \texttt{FLOAT\_TYPE'SAFE\_LARGE}$$

(b) When `X = 0.0`, see Section 11.

13.3. LOG — Logarithm to an Arbitrary Base

Declaration:

```
function LOG (X, BASE : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

$\texttt{LOG(X,BASE)} \approx \log_{\texttt{BASE}} \texttt{X}$

Usage:

```
Z := LOG(X, 10.0);    -- base 10 logarithm
Z := LOG(X, 2.0);     -- base 2 logarithm
Z := LOG(X, BASE);    -- base BASE logarithm
```

Domain:

(a) $X \geq 0.0$

(b) $BASE > 0.0$

(c) $BASE \neq 1.0$

Range:

Mathematically unbounded

Accuracy:

(a) Maximum relative error = `4.0` · `FLOAT_TYPE'BASE'EPSILON`

(b) `LOG(1.0, BASE) = 0.0`

Notes:

(a) When `BASE > 1.0`, the reachable range of `LOG` is approximately given by

$$\log_{\texttt{BASE}} \texttt{FLOAT\_TYPE'SAFE\_SMALL} \leq \texttt{LOG(X, BASE)} \leq \log_{\texttt{BASE}} \texttt{FLOAT\_TYPE'SAFE\_LARGE}$$

(b) When `0.0 < BASE < 1.0`, the reachable range of `LOG` is approximately given by

$$\log_{\texttt{BASE}} \texttt{FLOAT\_TYPE'SAFE\_LARGE} \leq \texttt{LOG(X, BASE)} \leq \log_{\texttt{BASE}} \texttt{FLOAT\_TYPE'SAFE\_SMALL}$$

(c) When `X = 0.0`, see Section 11.

13.4. EXP — Exponential Function

Declaration:

```
function EXP (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

`EXP(X)` $\approx e^{\texttt{X}}$

Usage:

```
Z := EXP(X);    -- e raised to the power X
```

Domain:

Mathematically unbounded

Range:

`EXP(X)` $\geq$ `0.0`

Accuracy:

(a) Maximum relative error = 4.0 · FLOAT_TYPE'BASE'EPSILON

(b) EXP(0.0) = 1.0

Notes:

The usable domain of EXP is approximately given by

$$X \leq \log_e \text{FLOAT\_TYPE'SAFE\_LARGE}$$

13.5. "**" — Exponentiation Operator

Declaration:

```
function "**" (X, Y : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

X ** Y $\approx X^Y$

Usage:

```
Z := X ** Y;    -- X raised to the power Y
```

Domain:

(a) X $\geq$ 0.0

(b) Y $\neq$ 0.0 when X = 0.0

Range:

X ** Y $\geq$ 0.0

Accuracy:

(a) Maximum relative error (when X > 0.0) =

$$\left(4.0 + \frac{|Y \cdot \log_e X|}{32.0}\right) \cdot \text{FLOAT\_TYPE'BASE'EPSILON}$$

(b) X ** 0.0 = 1.0 when X > 0.0

(c) 0.0 ** Y = 0.0 when Y > 0.0

(d) X ** 1.0 = X

(e) 1.0 ** Y = 1.0

Notes:

(a) The usable domain of "**", when X > 0.0, is approximately the set of values for X and Y satisfying

$$Y \cdot \log_e X \leq \log_e \texttt{FLOAT\_TYPE'SAFE\_LARGE}$$

This imposes a positive upper bound on Y (as a function of X) when X > 1.0, and a negative lower bound on Y (as a function of X) when 0.0 < X < 1.0.

(b) When X = 0.0 and Y < 0.0 (together), see Section 11.

13.6. SIN — Trigonometric Sine Function, Natural Cycle (Angle in Radians)

Declaration:

```
function SIN (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

SIN(X) $\approx \sin X$

Usage:

```
Z := SIN(X);    -- X in radians
```

Domain:

Mathematically unbounded

Range:

$|\texttt{SIN(X)}| \leq 1.0$

Accuracy:

(a) Maximum relative error = 2.0 · FLOAT_TYPE'BASE'EPSILON when |X| is less than or equal to some documented implementation-dependent threshold, which must not be less than

$$\texttt{FLOAT\_TYPE'MACHINE\_RADIX}^{\texttt{FLOAT\_TYPE'MACHINE\_MANTISSA}/2}$$

For larger values of |X|, degraded accuracy is allowed. An implementation must document its behavior for large |X|.

(b) SIN(0.0) = 0.0

13.7. SIN — Trigonometric Sine Function, Arbitrary Cycle (Angle in Arbitrary Units)

Declaration:

```
function SIN (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

SIN(X, CYCLE) $\approx \sin(2\pi \cdot \mathrm{X}/\mathrm{CYCLE})$

Usage:

```
Z := SIN(X, 360.0);    -- X in degrees
Z := SIN(X, CYCLE);    -- X in units such that one complete
                       -- cycle of rotation corresponds to
                       -- X = CYCLE
```

Domain:

(a) X mathematically unbounded

(b) CYCLE > 0.0

Range:

$|\mathrm{SIN(X, CYCLE)}| \leq 1.0$

Accuracy:

(a) Maximum relative error = 2.0 · FLOAT_TYPE'BASE'EPSILON

(b) For integer k, SIN(X, CYCLE) $= \begin{cases} 0.0, & \mathrm{X} = k \cdot \mathrm{CYCLE}/2.0 \\ 1.0, & \mathrm{X} = (4k+1) \cdot \mathrm{CYCLE}/4.0 \\ -1.0, & \mathrm{X} = (4k+3) \cdot \mathrm{CYCLE}/4.0 \end{cases}$

13.8. COS — Trigonometric Cosine Function, Natural Cycle (Angle in Radians)

Declaration:

```
function COS (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

COS(X) $\approx \cos X$

Usage:

```
Z := COS(X);    -- X in radians
```

Domain:

Mathematically unbounded

Range:

$|\text{COS(X)}| \leq 1.0$

Accuracy:

(a) Maximum relative error = 2.0 · FLOAT_TYPE'BASE'EPSILON when |X| is less than or equal to some documented implementation-dependent threshold, which must not be less than

$$\text{FLOAT\_TYPE'MACHINE\_RADIX}^{\text{FLOAT\_TYPE'MACHINE\_MANTISSA}/2}$$

For larger values of |X|, degraded accuracy is allowed. An implementation must document its behavior for large |X|.

(b) COS(0.0) = 1.0

13.9. COS — Trigonometric Cosine Function, Arbitrary Cycle (Angle in Arbitrary Units)

Declaration:

```
function COS (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

COS(X, CYCLE) $\approx \cos(2\pi \cdot \text{X/CYCLE})$

Usage:

```
Z := COS(X, 360.0);    -- X in degrees
Z := COS(X, CYCLE);    -- X in units such that one complete
                       -- cycle of rotation corresponds to
                       -- X = CYCLE
```

Domain:

(a) X mathematically unbounded

(b) CYCLE > 0.0

Range:

$|\text{COS(X, CYCLE)}| \leq 1.0$

Accuracy:

(a) Maximum relative error = 2.0 · FLOAT_TYPE'BASE'EPSILON

(b) For integer k, COS(X, CYCLE) $= \begin{cases} 1.0, & \text{X} = k \cdot \text{CYCLE} \\ 0.0, & \text{X} = (2k+1) \cdot \text{CYCLE}/4.0 \\ -1.0, & \text{X} = (2k+1) \cdot \text{CYCLE}/2.0 \end{cases}$

13.10. TAN — Trigonometric Tangent Function, Natural Cycle (Angle in Radians)

Declaration:

```
function TAN (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

TAN(X) $\approx \tan X$

Usage:

```
Z := TAN(X);    -- X in radians
```

Domain:

Mathematically unbounded

Range:

Mathematically unbounded

Accuracy:

(a) Maximum relative error = `4.0 · FLOAT_TYPE'BASE'EPSILON` when |`X`| is less than or equal to some documented implementation-dependent threshold, which must not be less than

$$\texttt{FLOAT\_TYPE'MACHINE\_RADIX}^{\texttt{FLOAT\_TYPE'MACHINE\_MANTISSA/2}}$$

For larger values of |`X`|, degraded accuracy is allowed. An implementation must document its behavior for large |`X`|.

(b) `TAN(0.0) = 0.0`

13.11. TAN — Trigonometric Tangent Function, Arbitrary Cycle (Angle in Arbitrary Units)

Declaration:

```
function TAN (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

`TAN(X, CYCLE)` $\approx \tan(2\pi \cdot$ `X/CYCLE`$)$

Usage:

```
Z := TAN(X, 360.0);    -- X in degrees
Z := TAN(X, CYCLE);    -- X in units such that one complete
                       -- cycle of rotation corresponds to
                       -- X = CYCLE
```

Domain:

(a) `X` mathematically unbounded

(b) `CYCLE > 0.0`

Range:

Mathematically unbounded

Accuracy:

(a) Maximum relative error = `4.0 · FLOAT_TYPE'BASE'EPSILON`

(b) `TAN(X, CYCLE) = 0.0` when `X` = $k \cdot$ `CYCLE/2.0`, for integer k

Notes:

When X = $(2k+1) \cdot$ CYCLE/4.0, for integer k, see Section 11.

13.12. COT — Trigonometric Cotangent Function, Natural Cycle (Angle in Radians)

Declaration:

```
function COT (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

COT(X) $\approx$ cot X

Usage:

```
Z := COT(X);    -- X in radians
```

Domain:

Mathematically unbounded

Range:

Mathematically unbounded

Accuracy:

Maximum relative error = 4.0 $\cdot$ FLOAT_TYPE'BASE'EPSILON when |X| is less than or equal to some documented implementation-dependent threshold, which must not be less than

$$\text{FLOAT\_TYPE'MACHINE\_RADIX}^{\text{FLOAT\_TYPE'MACHINE\_MANTISSA}/2}$$

For larger values of |X|, degraded accuracy is allowed. An implementation must document its behavior for large |X|.

Notes:

When X = 0.0, see Section 11.

13.13. COT — Trigonometric Cotangent Function, Arbitrary Cycle (Angle in Arbitrary Units)

Declaration:

```
function COT (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

COT(X, CYCLE) $\approx \cot(2\pi \cdot$ X/CYCLE$)$

Usage:

```
Z := COT(X, 360.0);    -- X in degrees
Z := COT(X, CYCLE);    -- X in units such that one complete
                       -- cycle of rotation corresponds to
                       -- X = CYCLE
```

Domain:

(a) X mathematically unbounded

(b) CYCLE $>$ 0.0

Range:

Mathematically unbounded

Accuracy:

(a) Maximum relative error = 4.0 $\cdot$ FLOAT_TYPE'BASE'EPSILON

(b) COT(X, CYCLE) = 0.0 when X $= (2k+1) \cdot$ CYCLE/4.0, for integer k

Notes:

When X $= k \cdot$ CYCLE/2.0, for integer k, see Section 11.

13.14. ARCSIN — Inverse Trigonometric Sine Function, Natural Cycle (Angle in Radians)

Declaration:

```
function ARCSIN (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

ARCSIN(X) $\approx$ arcsin X

Usage:

```
Z := ARCSIN(X);   -- Z in radians
```

Domain:

$|X| \leq 1.0$

Range:

$|ARCSIN(X)| \leq \pi/2$

Accuracy:

(a) Maximum relative error = 4.0 · FLOAT_TYPE'BASE'EPSILON
(b) ARCSIN(0.0) = 0.0
(c) ARCSIN(1.0) = $\pi/2$
(d) ARCSIN(-1.0) = $-\pi/2$

Notes:

$-\pi/2$ and $\pi/2$ are not safe numbers of FLOAT_TYPE. Accordingly, an implementation may exceed the range limits, but only slightly; cf. Section 9 for a precise statement of the requirements. Similarly, when accuracy requirement (c) or (d) applies, an implementation may approximate the prescribed result, but only within narrow limits; cf. Section 10 for a precise statement of the requirements.

13.15. ARCSIN — Inverse Trigonometric Sine Function, Arbitrary Cycle (Angle in Arbitrary Units)

Declaration:

```
function ARCSIN (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

ARCSIN(X, CYCLE) $\approx$ (arcsin X) · CYCLE/2π

Usage:

```
Z := ARCSIN(X, 360.0);   -- Z in degrees
Z := ARCSIN(X, CYCLE);   -- Z in units such that one complete
                         -- cycle of rotation corresponds to
                         -- Z = CYCLE
```

Domain:

(a) $|X| \leq 1.0$

(b) $CYCLE > 0.0$

Range:

$|ARCSIN(X, CYCLE)| \leq CYCLE/4.0$

Accuracy:

(a) Maximum relative error = $4.0 \cdot$ FLOAT_TYPE'BASE'EPSILON

(b) ARCSIN(0.0, CYCLE) = 0.0

(c) ARCSIN(1.0, CYCLE) = CYCLE/4.0

(d) ARCSIN(-1.0, CYCLE) = −CYCLE/4.0

Notes:

−CYCLE/4.0 and CYCLE/4.0 might not be safe numbers of FLOAT_TYPE. Accordingly, an implementation may exceed the range limits, but only slightly; cf. Section 9 for a precise statement of the requirements. Similarly, when accuracy requirement (c) or (d) applies, an implementation may approximate the prescribed result, but only within narrow limits; cf. Section 10 for a precise statement of the requirements.

13.16. ARCCOS — Inverse Trigonometric Cosine Function, Natural Cycle (Angle in Radians)

Declaration:

```
function ARCCOS (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

ARCCOS(X) $\approx \arccos X$

Usage:

```
Z := ARCCOS(X);   -- Z in radians
```

Domain:

$|X| \leq 1.0$

Range:

$0.0 \leq \text{ARCCOS}(X) \leq \pi$

Accuracy:

(a) Maximum relative error = 4.0 · FLOAT_TYPE'BASE'EPSILON

(b) ARCCOS(1.0) = 0.0

(c) ARCCOS(0.0) = $\pi/2$

(d) ARCCOS(-1.0) = π

Notes:

$\pi/2$ and π are not safe numbers of FLOAT_TYPE. Accordingly, an implementation may exceed the upper range limit, but only slightly; cf. Section 9 for a precise statement of the requirements. Similarly, when accuracy requirement (c) or (d) applies, an implementation may approximate the prescribed result, but only within narrow limits; cf. Section 10 for a precise statement of the requirements.

13.17. ARCCOS — Inverse Trigonometric Cosine Function, Arbitrary Cycle (Angle in Arbitrary Units)

Declaration:

```
function ARCCOS (X, CYCLE : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

ARCCOS(X, CYCLE) $\approx$ (arccos X) · CYCLE/2π

Usage:

```
Z := ARCCOS(X, 360.0);    -- Z in degrees
Z := ARCCOS(X, CYCLE);    -- Z in units such that one complete
                          -- cycle of rotation corresponds to
                          -- Z = CYCLE
```

Domain:

(a) $|X| \leq 1.0$

(b) CYCLE > 0.0

Range:

$0.0 \leq$ ARCCOS(X, CYCLE) $\leq$ CYCLE/2.0

Accuracy:

(a) Maximum relative error = 4.0 · FLOAT_TYPE'BASE'EPSILON

(b) ARCCOS(1.0, CYCLE) = 0.0

(c) ARCCOS(0.0, CYCLE) = CYCLE/4.0

(d) ARCCOS(-1.0, CYCLE) = CYCLE/2.0

Notes:

CYCLE/4.0 and CYCLE/2.0 might not be safe numbers of FLOAT_TYPE. Accordingly, an implementation may exceed the upper range limit, but only slightly; cf. Section 9 for a precise statement of the requirements. Similarly, when accuracy requirement (c) or (d) applies, an implementation may approximate the prescribed result, but only within narrow limits; cf. Section 10 for a precise statement of the requirements.

13.18. ARCTAN — Inverse Trigonometric Tangent Function, Natural Cycle (Angle in Radians)

Declaration:

```
function ARCTAN (Y : FLOAT_TYPE;
                 X : FLOAT_TYPE := 1.0) return FLOAT_TYPE;
```

Definition:

(a) ARCTAN(Y) $\approx \arctan$ Y

(b) ARCTAN(Y, X) $\approx \begin{cases} \arctan(Y/X), & X \geq 0.0 \\ (\arctan(Y/X)) + \pi, & X < 0.0 \text{ and } Y > 0.0 \\ \pm\pi \text{ (see note b)}, & X < 0.0 \text{ and } Y = 0.0 \\ (\arctan(Y/X)) - \pi, & X < 0.0 \text{ and } Y < 0.0 \end{cases}$

Usage:

```
Z := ARCTAN(Y);       -- Z, in radians, is the angle (in the
                      -- quadrant containing the point (1.0,Y))
                      -- whose tangent is Y
Z := ARCTAN(Y, X);    -- Z, in radians, is the angle (in the
                      -- quadrant containing the point (X,Y))
                      -- whose tangent is Y/X
```

Domain:

$\texttt{X} \neq \texttt{0.0}$ when $\texttt{Y} = \texttt{0.0}$

Range:

(a) $|\texttt{ARCTAN(Y)}| \leq \pi/2$

(b) $\texttt{0.0} \leq \texttt{ARCTAN(Y, X)} \leq \pi$ when $\texttt{Y} \geq \texttt{0.0}$

(c) $-\pi \leq \texttt{ARCTAN(Y, X)} \leq \texttt{0.0}$ when $\texttt{Y} \leq \texttt{0.0}$ (see note b regarding the overlap at $\texttt{Y} = \texttt{0.0}$)

Accuracy:

(a) Maximum relative error = $\texttt{4.0} \cdot \texttt{FLOAT\_TYPE'BASE'EPSILON}$

(b) $\texttt{ARCTAN(0.0)} = \texttt{0.0}$

(c) $\texttt{ARCTAN(0.0, X)} = \begin{cases} \texttt{0.0}, & \texttt{X} > \texttt{0.0} \\ \pm\pi \text{ (see note b)}, & \texttt{X} < \texttt{0.0} \end{cases}$

(d) $\texttt{ARCTAN(Y, 0.0)} = \begin{cases} \pi/2, & \texttt{Y} > \texttt{0.0} \\ -\pi/2, & \texttt{Y} < \texttt{0.0} \end{cases}$

Notes:

(a) $-\pi$, $-\pi/2$, $\pi/2$, and π are not safe numbers of FLOAT_TYPE. Accordingly, an implementation may exceed the range limits that involve these numbers, but only slightly; cf. Section 9 for a precise statement of the requirements. Similarly, when accuracy requirement (c) or (d) applies, an implementation may approximate a prescribed result that involves one of these numbers, but only within narrow limits; cf. Section 10 for a precise statement of the requirements.

(b) Two cases arise when $\texttt{X} < \texttt{0.0}$ and $\texttt{Y} = \texttt{0.0}$:

  (i) in an implementation exploiting signed zeros (cf. Section 4), ARCTAN must deliver $-\pi$ when Y is a negatively signed zero and π when Y is a positively signed zero;

  (ii) in an implementation not exploiting signed zeros, ARCTAN must deliver π.

13.19. ARCTAN — Inverse Trigonometric Tangent Function, Arbitrary Cycle (Angle in Arbitrary Units)

Declaration:

```
function ARCTAN (Y     : FLOAT_TYPE;
                 X     : FLOAT_TYPE := 1.0;
                 CYCLE : FLOAT_TYPE)        return FLOAT_TYPE;
```

Definition:

(a) `ARCTAN(Y, CYCLE => CYCLE)` $\approx (\arctan \texttt{Y}) \cdot \texttt{CYCLE}/2\pi$

(b) `ARCTAN(Y, X, CYCLE)` $\approx \begin{cases} (\arctan(\texttt{Y}/\texttt{X})) \cdot \texttt{CYCLE}/2\pi, & \texttt{X} \geq 0.0 \\ ((\arctan(\texttt{Y}/\texttt{X})) + \pi) \cdot \texttt{CYCLE}/2\pi, & \texttt{X} < 0.0 \text{ and } \texttt{Y} > 0.0 \\ \pm\texttt{CYCLE}/2.0 \text{ (see note b)}, & \texttt{X} < 0.0 \text{ and } \texttt{Y} = 0.0 \\ ((\arctan(\texttt{Y}/\texttt{X})) - \pi) \cdot \texttt{CYCLE}/2\pi, & \texttt{X} < 0.0 \text{ and } \texttt{Y} < 0.0 \end{cases}$

Usage:

```
Z := ARCTAN(Y, CYCLE => 360.0);    -- Z, in degrees, is the
                                   -- angle (in the quadrant
                                   -- containing the point
                                   -- (1.0,Y)) whose tangent is Y
Z := ARCTAN(Y, CYCLE => CYCLE);    -- Z, in units such that one
                                   -- complete cycle of rotation
                                   -- corresponds to Z = CYCLE,
                                   -- is the angle (in the
                                   -- quadrant containing the
                                   -- point (1.0,Y)) whose
                                   -- tangent is Y
Z := ARCTAN(Y, X, 360.0);          -- Z, in degrees, is the
                                   -- angle (in the quadrant
                                   -- containing the point (X,Y))
                                   -- whose tangent is Y/X
Z := ARCTAN(Y, X, CYCLE);          -- Z, in units such that one
                                   -- complete cycle of rotation
                                   -- corresponds to Z = CYCLE,
                                   -- is the angle (in the
                                   -- quadrant containing the
                                   -- point (X,Y)) whose
                                   -- tangent is Y/X
```

Domain:

(a) `X` $\neq$ `0.0` when `Y` $=$ `0.0`

(b) `CYCLE` $>$ `0.0`

Range:

(a) $|$`ARCTAN(Y, CYCLE => CYCLE)`$| \leq$ `CYCLE/4.0`

(b) `0.0` $\leq$ `ARCTAN(Y, X, CYCLE)` $\leq$ `CYCLE/2.0` when `Y` $\geq$ `0.0`

(c) $-$`CYCLE/2.0` $\leq$ `ARCTAN(Y, X, CYCLE)` $\leq$ `0.0` when `Y` $\leq$ `0.0` (see note b regarding the overlap at `Y` $=$ `0.0`)

Accuracy:

(a) Maximum relative error = `4.0` · `FLOAT_TYPE'BASE'EPSILON`

(b) `ARCTAN(0.0, CYCLE => CYCLE) = 0.0`

(c) `ARCTAN(0.0, X, CYCLE)` = $\begin{cases} 0.0, & X > 0.0 \\ \pm \texttt{CYCLE/2.0} \text{ (see note b)}, & X < 0.0 \end{cases}$

(d) `ARCTAN(Y, 0.0, CYCLE)` = $\begin{cases} \texttt{CYCLE/4.0}, & Y > 0.0 \\ -\texttt{CYCLE/4.0}, & Y < 0.0 \end{cases}$

Notes:

(a) `-CYCLE/2.0`, `-CYCLE/4.0`, `CYCLE/4.0`, and `CYCLE/2.0` might not be safe numbers of `FLOAT_TYPE`. Accordingly, an implementation may exceed the range limits that involve these numbers, but only slightly; cf. Section 9 for a precise statement of the requirements. Similarly, when accuracy requirement (c) or (d) applies, an implementation may approximate a prescribed result that involves one of these numbers, but only within narrow limits; cf. Section 10 for a precise statement of the requirements.

(b) Two cases arise when `X < 0.0` and `Y = 0.0`:

(i) in an implementation exploiting signed zeros (cf. Section 4), `ARCTAN` must deliver `-CYCLE/2.0` when `Y` is a negatively signed zero and `CYCLE/2.0` when `Y` is a positively signed zero;

(ii) in an implementation not exploiting signed zeros, `ARCTAN` must deliver `CYCLE/2.0`.

13.20. ARCCOT — Inverse Trigonometric Cotangent Function, Natural Cycle (Angle in Radians)

Declaration:

```
function ARCCOT (X : FLOAT_TYPE;
                 Y : FLOAT_TYPE := 1.0) return FLOAT_TYPE;
```

Definition:

(a) `ARCCOT(X)` ≈ arccot `X`

(b) `ARCCOT(X, Y)` ≈ $\begin{cases} \text{arccot}(X/Y), & Y > 0.0 \\ 0.0, & Y = 0.0 \text{ and } X > 0.0 \\ \pm\pi \text{ (see note b)}, & Y = 0.0 \text{ and } X < 0.0 \\ (\text{arccot}(X/Y)) - \pi, & Y < 0.0 \end{cases}$

Usage:

```
Z := ARCCOT(X);       -- Z, in radians, is the angle (in the
                      -- quadrant containing the point (X,1.0)
                      -- whose cotangent is X
Z := ARCCOT(X, Y);    -- Z, in radians, is the angle (in the
                      -- quadrant containing the point (X,Y))
                      -- whose cotangent is X/Y
```

Domain:

$\texttt{Y} \neq \texttt{0.0}$ when $\texttt{X} = \texttt{0.0}$

Range:

(a) $\texttt{0.0} \leq \texttt{ARCCOT(X)} \leq \pi$

(b) $\texttt{0.0} \leq \texttt{ARCCOT(X, Y)} \leq \pi$ when $\texttt{Y} \geq \texttt{0.0}$

(c) $-\pi \leq \texttt{ARCCOT(X, Y)} \leq \texttt{0.0}$ when $\texttt{Y} \leq \texttt{0.0}$ (see note b regarding the overlap at $\texttt{Y} = \texttt{0.0}$)

Accuracy:

(a) Maximum relative error = $\texttt{4.0} \cdot \texttt{FLOAT\_TYPE'BASE'EPSILON}$

(b) $\texttt{ARCCOT(0.0)} = \pi/2$

(c) $\texttt{ARCCOT(0.0, Y)} = \begin{cases} \pi/2, & \texttt{Y} > \texttt{0.0} \\ -\pi/2, & \texttt{Y} < \texttt{0.0} \end{cases}$

(d) $\texttt{ARCCOT(X, 0.0)} = \begin{cases} \texttt{0.0}, & \texttt{X} > \texttt{0.0} \\ \pm\pi \text{ (see note b)}, & \texttt{X} < \texttt{0.0} \end{cases}$

Notes:

(a) $-\pi$, $-\pi/2$, $\pi/2$, and π are not safe numbers of FLOAT_TYPE. Accordingly, an implementation may exceed the range limits that involve these numbers, but only slightly; cf. Section 9 for a precise statement of the requirements. Similarly, when accuracy requirement (b), (c), or (d) applies, an implementation may approximate a prescribed result that involves one of these numbers, but only within narrow limits; cf. Section 10 for a precise statement of the requirements.

(b) Two cases arise when $\texttt{X} < \texttt{0.0}$ and $\texttt{Y} = \texttt{0.0}$:

  (i) in an implementation exploiting signed zeros (cf. Section 4), ARCCOT must deliver $-\pi$ when Y is a negatively signed zero and π when Y is a positively signed zero;

  (ii) in an implementation not exploiting signed zeros, ARCCOT must deliver π.

13.21. ARCCOT — Inverse Trigonometric Cotangent Function, Arbitrary Cycle (Angle in Arbitrary Units)

Declaration:

```
function ARCCOT (X     : FLOAT_TYPE;
                 Y     : FLOAT_TYPE := 1.0;
                 CYCLE : FLOAT_TYPE)        return FLOAT_TYPE;
```

Definition:

(a) ARCCOT(X, CYCLE => CYCLE) $\approx$ (arccot X) $\cdot$ CYCLE/2π

(b) ARCCOT(X, Y, CYCLE) $\approx$
$$\begin{cases} (\text{arccot}\,(X/Y)) \cdot \text{CYCLE}/2\pi, & Y > 0.0 \\ 0.0, & Y = 0.0 \text{ and } X > 0.0 \\ \pm\text{CYCLE}/2.0 \text{ (see note b)}, & Y = 0.0 \text{ and } X < 0.0 \\ ((\text{arccot}\,(X/Y)) - \pi) \cdot \text{CYCLE}/2\pi, & Y < 0.0 \end{cases}$$

Usage:

```
Z := ARCCOT(X, CYCLE => 360.0);   -- Z, in degrees, is the
                                  -- angle (in the quadrant
                                  -- containing the point
                                  -- (X,1.0)) whose cotangent
                                  -- is X
Z := ARCCOT(X, CYCLE => CYCLE);   -- Z, in units such that one
                                  -- complete cycle of rotation
                                  -- corresponds to Z = CYCLE,
                                  -- is the angle (in the
                                  -- quadrant containing the
                                  -- point (X,1.0)) whose
                                  -- cotangent is X
Z := ARCCOT(X, Y, 360.0);         -- Z, in degrees, is the
                                  -- angle (in the quadrant
                                  -- containing the point (X,Y))
                                  -- whose cotangent is X/Y
Z := ARCCOT(X, Y, CYCLE);         -- Z, in units such that one
                                  -- complete cycle of rotation
                                  -- corresponds to Z = CYCLE
                                  -- is the angle (in the
                                  -- quadrant containing the
                                  -- point (X,Y)) whose
                                  -- cotangent is X/Y
```

Domain:

(a) Y $\neq$ 0.0 when X = 0.0

(b) CYCLE > 0.0

Range:

(a) 0.0 $\leq$ ARCCOT(X, CYCLE => CYCLE) $\leq$ CYCLE/2.0

(b) 0.0 $\leq$ ARCCOT(X, Y, CYCLE) $\leq$ CYCLE/2.0 when Y $\geq$ 0.0

(c) $-$CYCLE/2.0 $\leq$ ARCCOT(X, Y, CYCLE) $\leq$ 0.0 when Y $\leq$ 0.0 (see note b regarding the overlap at Y = 0.0)

Accuracy:

(a) Maximum relative error = `4.0` · `FLOAT_TYPE'BASE'EPSILON`

(b) `ARCCOT(0.0, CYCLE => CYCLE)` = `CYCLE/4.0`

(c) `ARCCOT(0.0, Y, CYCLE)` = $\begin{cases} \texttt{CYCLE/4.0}, & \texttt{Y} > \texttt{0.0} \\ \texttt{-CYCLE/4.0}, & \texttt{Y} < \texttt{0.0} \end{cases}$

(d) `ARCCOT(X, 0.0, CYCLE)` = $\begin{cases} \texttt{0.0}, & \texttt{X} > \texttt{0.0} \\ \pm\texttt{CYCLE/2.0} \text{ (see note b)}, & \texttt{X} < \texttt{0.0} \end{cases}$

Notes:

(a) `-CYCLE/2.0`, `-CYCLE/4.0`, `CYCLE/4.0`, and `CYCLE/2.0` might not be safe numbers of `FLOAT_TYPE`. Accordingly, an implementation may exceed the range limits that involve these numbers, but only slightly; cf. Section 9 for a precise statement of the requirements. Similarly, when accuracy requirement (b), (c), or (d) applies, an implementation may approximate a prescribed result that involves one of these numbers, but only within narrow limits; cf. Section 10 for a precise statement of the requirements.

(b) Two cases arise when `X` < `0.0` and `Y` = `0.0`:

(i) in an implementation exploiting signed zeros (cf. Section 4), `ARCCOT` must deliver `-CYCLE/2.0` when `Y` is a negatively signed zero and `CYCLE/2.0` when `Y` is a positively signed zero;

(ii) in an implementation not exploiting signed zeros, `ARCCOT` must deliver `CYCLE/2.0`.

13.22. SINH — Hyperbolic Sine Function

Declaration:

```
function SINH (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

`SINH(X)` ≈ sinh `X`

Usage:

```
Z := SINH(X);
```

Domain:

Mathematically unbounded

Range:

Mathematically unbounded

Accuracy:

(a) Maximum relative error = `8.0 · FLOAT_TYPE'BASE'EPSILON`
(b) `SINH(0.0) = 0.0`

Notes:

The usable domain of `SINH` is approximately given by

$$|\mathtt{X}| \leq \log_e \mathtt{FLOAT\_TYPE'SAFE\_LARGE} + \log_e \mathtt{2.0}$$

13.23. COSH — Hyperbolic Cosine Function

Declaration:

```
function COSH (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

`COSH(X)` $\approx \cosh$ `X`

Usage:

```
Z := COSH(X);
```

Domain:

Mathematically unbounded

Range:

`COSH(X)` $\geq$ `1.0`

Accuracy:

(a) Maximum relative error = `8.0 · FLOAT_TYPE'BASE'EPSILON`
(b) `COSH(0.0) = 1.0`

Notes:

The usable domain of `COSH` is approximately given by

$$|\mathtt{X}| \leq \log_e \mathtt{FLOAT\_TYPE'SAFE\_LARGE} + \log_e \mathtt{2.0}$$

13.24. TANH — Hyperbolic Tangent Function

Declaration:

```
function TANH (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

TANH(X) $\approx \tanh$ X

Usage:

```
Z := TANH(X);
```

Domain:

Mathematically unbounded

Range:

$|$TANH(X)$| \leq$ 1.0

Accuracy:

(a) Maximum relative error = 8.0 · FLOAT_TYPE'BASE'EPSILON
(b) TANH(0.0) = 0.0

13.25. COTH — Hyperbolic Cotangent Function

Declaration:

```
function COTH (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

COTH(X) $\approx \coth$ X

Usage:

```
Z := COTH(X);
```

Domain:

Mathematically unbounded

Range:

$|\texttt{COTH(X)}| \geq 1.0$

Accuracy:

Maximum relative error = `8.0 * FLOAT_TYPE'BASE'EPSILON`

Notes:

When `X = 0.0`, see Section 11.

13.26. ARCSINH — Inverse Hyperbolic Sine Function

Declaration:

```
function ARCSINH (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

`ARCSINH(X)` $\approx$ arcsinh `X`

Usage:

```
Z := ARCSINH(X);
```

Domain:

Mathematically unbounded

Range:

Mathematically unbounded

Accuracy:

(a) Maximum relative error = `8.0 * FLOAT_TYPE'BASE'EPSILON`

(b) `ARCSINH(0.0) = 0.0`

Notes:

The reachable range of ARCSINH is approximately given by

$$|\texttt{ARCSINH(X)}| \leq \log_e \texttt{FLOAT\_TYPE'SAFE\_LARGE} + \log_e \texttt{2.0}$$

13.27. ARCCOSH — Inverse Hyperbolic Cosine Function

Declaration:

```
function ARCCOSH (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

ARCCOSH(X) $\approx$ arccosh X

Usage:

```
Z := ARCCOSH(X);
```

Domain:

X $\geq$ 1.0

Range:

ARCCOSH(X) $\geq$ 0.0

Accuracy:

(a) Maximum relative error = 8.0 · FLOAT_TYPE'BASE'EPSILON
(b) ARCCOSH(1.0) = 0.0

Notes:

The upper bound of the reachable range of ARCCOSH is approximately given by

$$\texttt{ARCCOSH(X)} \leq \log_e \texttt{FLOAT\_TYPE'SAFE\_LARGE} + \log_e \texttt{2.0}$$

13.28. ARCTANH — Inverse Hyperbolic Tangent Function

Declaration:

```
function ARCTANH (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

ARCTANH(X) $\approx$ arctanh X

Usage:

```
Z := ARCTANH(X);
```

Domain:

$|X| \leq 1.0$

Range:

Mathematically unbounded

Accuracy:

(a) Maximum relative error = 8.0 · FLOAT_TYPE'BASE'EPSILON
(b) ARCTANH(0.0) = 0.0

Notes:

When X = $\pm$1.0, see Section 11.

13.29. ARCCOTH — Inverse Hyperbolic Cotangent Function

Declaration:

```
function ARCCOTH (X : FLOAT_TYPE) return FLOAT_TYPE;
```

Definition:

ARCCOTH(X) $\approx$ arccoth X

Usage:

```
Z := ARCCOTH(X);
```

Domain:

$|X| \geq 1.0$

Range:

Mathematically unbounded

Accuracy:

Maximum relative error = `8.0 · FLOAT_TYPE'BASE'EPSILON`

Notes:

When `X` = ±`1.0`, see Section 11.

Rationale for the Proposed Standard for a Generic Package of Elementary Functions for Ada

Kenneth W. Dritz

December 1990

Abstract

This paper supplements the "Proposed Standard for a Generic Package of Elementary Functions for Ada," written by the ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group. Based on recommendations made jointly by the ACM SIGAda Numerics Working Group and the Ada-Europe Numerics Working Group, the proposed elementary functions standard is the first of several anticipated collateral standards to address the interrelated issues of portability, efficiency, and robustness of numerical software written in Ada. Organized as a series of questions and answers, this supplement outlines the reasoning by which the proposed standard came to acquire certain features and exclude others.

Introduction

In the four years since the ACM SIGAda Numerics Working Group first began to work with the Ada-Europe Numerics Working Group on a standard specification for a generic package of elementary functions for Ada, it has received numerous inquiries about details of that specification from observers of the effort and from potential future implementors. Particular questions—especially one about the handling of certain optional parameters—have been answered more than once. This paper (a revision of [6]) has been written not just to provide ready answers to those questions which, by their demonstrated popularity, we can expect to be asked again; it tries to anticipate others, and it collects in one place—as a kind of separate appendix to the proposed standard—the sometimes subtle reasons why certain decisions were reached during the development of the standard. With the latter, we include the reasons for deciding *not* to do things in certain obvious ways. This paper also discusses the less readily apparent implications of some of those decisions for implementors. Lastly, we hope that it will enhance the understanding of the proposed standard and facilitate its review.

Why is a standard needed, and how did this proposed standard come to be?

The absence of predefined elementary functions from Ada has been one of the deterrents to the portability of scientific and engineering applications software written in that language. The need for such functions has been widely recognized, as evidenced by the support given to them by compiler vendors in the form of proprietary packages, as well as by several purveyors of libraries of mathematical software. While this has served the immediate needs of programmers within their own environments, it has done little to solve the broader problem of portability of applications software using the elementary functions. The reason, of course, is the lack of commonality among the different packages: they differ in the number of functions implemented, their names, their parameter profiles, the handling of exceptional conditions, and even the use (or avoidance) of genericity.

Instead of including predefined elementary functions in Ada, its authors gave the language the necessary general features for defining and collecting subprograms together into libraries (e.g., packages, generics, and subprogram overloading), and for creating portable and efficient numerical software in particular (e.g., a model of floating-point arithmetic, and environmental enquiries in the form of attributes), and then left it to experienced numerical analysts to do what they are uniquely qualified to do: apply those features to the task of specifying and implementing high-quality libraries of mathematical software. Numerical analysts were already engaged in this work before Ada itself was standardized. The need for standards was recognized early, with several preliminary proposals [9, 21,

20] published between 1982 and 1987. Other seminal papers on the content, philosophy, and implementation of scientific libraries in Ada were collected together in [10] in 1986.

People and papers came together beginning in about that same year to form committees with working documents. The standardization effort has been supported and encouraged in the United States by the Ada Joint Program Office of the U. S. Department of Defense, and in Europe by the Commission of the European Communities. The ACM SIGAda Numerics Working Group and the Ada-Europe Numerics Working Group have worked together closely since then. Interim reports on the work of the former committee were presented in 1987 at the International Conference on the Ada Programming Language [22], in 1988 at the Sandia Workshop on Ada in Real-Time and Scientific Environments [7, 26], and more recently in several tutorials and colloquia. Various drafts of the working papers that eventually resulted in the present proposal were circulated, and the response to them has been enthusiastic. This work was adopted by the WG9 Numerics Rapporteur Group [15] in March of 1989 and presented to WG9 as a proposed standard. WG9 approved the proposal in June, 1989, subject to minor revisions. The revised proposal [16] was initially processed by SC22 in early 1990, leading to a few editorial changes. The SIGAda Numerics Working Group took advantage of that opportunity for change to incorporate a minor technical improvement, which permits implementations having the capability of exploiting signed zeros (a feature of the IEEE standards for floating-point arithmetic [12, 13]) to do so in appropriate ways. At the same time, the Ada-Europe Numerics Working Group advocated a change stemming from the difficulty that some implementors encountered while trying to produce efficient implementations that are also portable. The second revision of the proposal [17], endorsed by the Numerics Rapporteur Group in December 1990, is the document that this rationale describes.

The rest of this paper is intended to be read in conjunction with the proposal.

Why does the proposed standard define a generic package?

The package construct is the obvious mechanism for encapsulating a functionally cohesive set of subprograms and their related exceptions, global data, etc. The facilities of `TEXT_IO` are made available through that mechanism, for example. Using a generic package instead of an ordinary package is appropriate, furthermore, when the facilities to be encapsulated need to be parameterized by some property of the application in which they are to be used. In view of the rules for parameter associations, the inability to anticipate which floating-point type (or types) the programmer will choose for the application dictates that the package containing the elementary functions be made generic on the type of their formal parameters and returned value.

For that reason the elementary functions package is indeed generic. It is named `GENERIC_ELEMENTARY_FUNCTIONS`, and it has one generic formal parameter, which is a generic formal type named `FLOAT_TYPE`. An instantiation of the generic package with a generic actual type (which can be any floating-point subtype) produces a package containing elementary functions that can be invoked with an argument or arguments of that subtype.

Can the generic actual type contain a range constraint?

Until November 1990, the answer to this question was yes, but at that time the proposed standard was changed to make the answer dependent on the implementation. That is, implementations are allowed to impose a restriction that the generic actual type must not contain a range constraint that reduces the range of allowable values; alternatively, they may permit such range constraints (but then they have an obligation to prevent a range constraint in the generic actual type from interfering with their ability to deliver a value, which constrains the available implementation strategies somewhat). So that users will know what to expect from any particular implementation, such a restriction must be documented if it is imposed.

This issue was debated at the time that the earliest draft of the proposed standard was being formulated. The potential difficulty (for implementors) of allowing a range constraint in the generic actual type was recognized immediately. Since the role of the generic formal type is essentially to parameterize the "working precision" in the generic package, it is tempting for implementors to use the generic formal type, `FLOAT_TYPE`, as the type mark in the declarations of temporary variables and perhaps even of constants in the body of `GENERIC_ELEMENTARY_FUNCTIONS`. The obvious problem with this straightforward implementation strategy is that `FLOAT_TYPE` inherits any range constraint that the corresponding generic actual type happens to contain, and this could well invalidate assignments to temporary variables and initialization of constants in the body—even when the argument in a particular

function invocation, as well as the final result (if it could only be computed), satisfies the range constraint. The user must accept the responsibility of being able to pass values into and out of an elementary function whose argument and result type are range constrained, of course, since constraint checks are required by Ada in those contexts and nothing the implementor does in the body can avoid them. So, for example, if the user instantiates GENERIC_ELEMENTARY_FUNCTIONS with a type declared as "digits 6 range 3.0 .. 20.0," then there is no hope of asking for the square root of 25.0 (because the argument will be outside the range of SQRT's parameter type) or of 4.0 (because the result will be outside the range of SQRT's result type), and any attempt to do so must necessarily raise CONSTRAINT_ERROR. On the other hand, it was universally judged to be unacceptable for an implementation to raise CONSTRAINT_ERROR when the square root of, say, 16.0 is requested (with the same range-constrained generic actual type), since both the argument and the result are within the range of the parameter and result types. And yet, there is a good chance that a straightforward implementation—one that uses variables or constants of type FLOAT_TYPE—*will* raise CONSTRAINT_ERROR in this case and in other seemingly innocuous cases.[1] Thus, in agreeing to allow range constraints in the generic actual type, the committee made it clear in the earliest drafts that such "gratuitous" exceptions must be avoided by implementations. It did so only after concluding that suitable implementation techniques (that avoided the use of FLOAT_TYPE as a type mark in declarations) were available.

The committee strongly favored allowing range constraints, at first. It felt that users would not accept having their freedom to instantiate GENERIC_ELEMENTARY_FUNCTIONS with *any* floating-point type taken away. Indeed, earlier versions of this rationale declaimed that the interests of users outweighed those of implementors in settling the range constraints issue.

The problems facing implementors who wish to allow range constraints would vanish if there were a way, in a generic body, of declaring a variable with the precision of a floating-point generic formal parameter but without its range constraints, if any. Declaring a variable to be of type "digits FLOAT_TYPE'BASE'DIGITS" will not suffice, because the expression in a floating accuracy definition is required to be static, and an attribute of a generic formal parameter is not static. Declaring a variable to be of type "FLOAT_TYPE'BASE" comes to mind also, but this is invalid because the BASE attribute can be used only in a prefix for other attributes.

What implementation strategies *are* available to implementors who wish to allow range constraints?

One method is to represent each elementary function, at the highest level, by a "shell" that merely "dispatches" to a lower-level function supporting the required precision. The proper lower-level function is determined by querying FLOAT_TYPE'BASE'DIGITS in a case-statement whose choices test symbolically for membership in the range of precisions of each of the available predefined floating-point types. Lower-level versions of each of the elementary functions can be provided for each predefined floating-point type by one instantiation of an inner generic package for each such type. Because the generic actual type used to instantiate this inner generic package is never range constrained, the inner generic package of lower-level functions can use its generic formal parameter for the type of its working variables without fear of violating range constraints. A straightforward and often-used strategy, this method has two drawbacks:

- If the method is to be portable, the inner generic package must be designed to accommodate, in each elementary function, the entire range of precisions to be supported. The multiple instantiations of this inner package will then lead, with some Ada compilers, to multiple copies of the code applicable to a given precision, even though only one copy is logically required. Thus, for example, code to perform double-precision computations will be present in the instantiation for a double-precision predefined type as well as in the instantiation for a single-precision predefined type, even though the latter will never be called upon to perform those computations. Currently implemented optimizations do not, in general, attack this version of the dead-code removal problem.
- The method suffers from a lack of portability related to variations, from one implementation of Ada to the next, in the names and number of the available predefined floating-point types. As a supplement to this strategy, a technique of Chebat [4] can be used to extend portability to a fixed set of *potentially* predefined type names chosen by the implementor, even if some names in the set do not actually exist as predefined types of the Ada implementation; however, Chebat's technique will not pick up predefined type names outside the anticipated set.

[1] One standard argument reduction strategy is to transform the argument to a value in the range 0.25 .. 1.0. Clearly, an attempt to store the transformed argument in a variable of type FLOAT_TYPE will violate the inherited range constraint.

A second method, which completely avoids reliance on predefined type names, has a similar case-statement in the body of each elementary function, with the choices represented by constants and with each case containing a block-statement that declares working variables of the same precision, given as a constant. In practice, the number of cases can be sharply reduced, as shown by Tang in [25], by grouping a range of successive precision values together into each choice, with the breakpoints chosen with representative hardware in mind. (The precision used in the declaration of working variables in each case then becomes the constant representing the upper bound of the range of precisions of the case's choice. Attention to several details not discussed here is required.) This method, too, has two drawbacks:

- Excess precision may sometimes be used when not required—i.e., precision may be wasted—in order to keep the cases to a manageable number.
- Expensive compromises are required to fit some of the steps of the typical realization of an elementary function into the case structure. The approximation step—which is sandwiched between the argument reduction step and the result construction step—fits into the case structure well; each case not only provides the precision to be used in the declarations of its local working variables, but it also serves as the locus within which the chosen approximation scheme can be tailored to that precision (for example, by using the appropriate number of terms, with appropriate coefficients, in a polynomial). On the other hand, the dependence of the argument reduction and result construction steps on the precision required is confined to the declarations of the working variables needed in those steps; the same argument reduction algorithm or result construction algorithm can be used in each of the cases. To avoid the needless duplication of code (differing only in the precision of variables), one is motivated to take the argument reduction and result construction steps out of all the cases and to place a common argument reduction step before the case-statement and a common result construction step after the case-statement. The only precision that suffices for the variables used in these steps is the maximum available precision. When that precision is more than is required (e.g., because a case corresponding to a low precision is selected), this tactic can produce an unfortunate performance degradation, especially in Ada implementations in which the highest precision available is simulated in software.

The magnitude of the performance penalty induced by the second method, which nevertheless is favored by some implementors because it is inherently more portable than the first, became apparent only recently, as implementation experience began to build. Aided by a growing suspicion that the desire to instantiate `GENERIC_ELEMENTARY_FUNCTIONS` with a range-constrained generic actual type may not be as realistic as once thought,[2] the realization led to a reevaluation of the original decision on range constraints, and to a recommendation from the Ada-Europe Numerics Working Group that the decision be partially reversed by allowing an implementation to impose a restriction against range constraints if it so wishes. By imposing this restriction, an implementation can use `FLOAT_TYPE` directly as the type mark for all of its working variables; the only need for a case-statement is to select an approximation method tailored to the precision required, and the cases will furthermore not need to contain block-statements.

A programmer who requires portability to all implementations must avoid instantiating `GENERIC_ELEMENTARY_FUNCTIONS` with a range-constrained generic actual type. That is an inconvenience, but it is hardly more than that. A program designed around a range-constrained application type or subtype named `APPLICATION_TYPE`, used both to declare variables and to instantiate a generic math package, can be systematically modified to avoid instantiating `GENERIC_ELEMENTARY_FUNCTIONS` with a range-constrained generic actual type as follows:

- Find the declaration of the type or subtype `APPLICATION_TYPE`.
- Change the declaration so that the type or subtype has a new name, say `ORIGINAL_TYPE`. In the modified program, `ORIGINAL_TYPE` will be used *only* as the source of properties for a new type, `BASE_TYPE`, and a subtype thereof called `APPLICATION_TYPE`.
- Declare the type `BASE_TYPE` as "`digits ORIGINAL_TYPE'BASE'DIGITS`." Instantiate `GENERIC_ELEMENTARY_FUNCTIONS` with `BASE_TYPE` instead of `APPLICATION_TYPE`.
- Declare the subtype `APPLICATION_TYPE` as "`BASE_TYPE range BASE_TYPE(ORIGINAL_TYPE'FIRST) .. BASE_TYPE(ORIGINAL_TYPE'LAST)`." Use it in the same ways as in the original program, except for the instantiation of `GENERIC_ELEMENTARY_FUNCTIONS`.

[2] It is hard to imagine how a *single* application-determined range constraint can be suitably applied to the inputs and outputs of *all* the elementary functions without causing a constraint violation *somewhere* on a call or a return.

The foregoing prescription caters to the worst case, in which the base type of APPLICATION_TYPE in the original program is anonymous; obviously, if its name is known, that name can simply be used to instantiate GENERIC_ELEMENTARY_FUNCTIONS. Also, the foregoing prescription assumes that APPLICATION_TYPE in the original program is not a generic formal type; if it is, the same technique can be applied one level out.

The proposed standard also implies that implementations imposing the restriction must behave predictably when the restriction is violated; it says that instantiation of GENERIC_ELEMENTARY_FUNCTIONS with a generic actual type containing a range constraint that reduces the allowable range of values, in violation of the restriction, must result either in the rejection of the compilation of a unit containing an instantiation of GENERIC_ELEMENTARY_FUNCTIONS or in the raising of CONSTRAINT_ERROR or PROGRAM_ERROR during the elaboration of an instantiation of GENERIC_ELEMENTARY_FUNCTIONS. The first two actions are consequences of the semantics of Ada and are beyond the implementor's ability to influence; if neither of these occurs first, the implementor can ensure that the last action takes place by coding the following in the statement-sequence of the body of GENERIC_ELEMENTARY_FUNCTIONS:

```
if FLOAT_TYPE'FIRST > FLOAT_TYPE'BASE'FIRST or
   FLOAT_TYPE'LAST  < FLOAT_TYPE'BASE'LAST  then
   raise PROGRAM_ERROR;
end if;
```

Finally, since the Ada 9X Requirements Team has already recognized the need for improvements in the treatment of generics that would eliminate the problems faced by implementors wishing to allow range constraints in the generic actual type ([1], User Need U4.4-A), it can be anticipated that there will one day be no reason to allow implementations to impose a restriction against range constraints; at that time, the GENERIC_ELEMENTARY_FUNCTIONS standard will be revised to revoke the right of an implementation to impose such a restriction.

What functions are included?

Nineteen functions and one operator are defined in the elementary functions package. These are SQRT; the two functions (EXP and LOG) and one operator ("**") of the exponential family; the four commonly encountered functions (SIN, COS, TAN, and COT) of the trigonometric family; their inverses (ARCSIN, ARCCOS, ARCTAN, and ARCCOT); the four commonly encountered functions (SINH, COSH, TANH, and COTH) of the hyperbolic family; and their inverses (ARCSINH, ARCCOSH, ARCTANH, and ARCCOTH). They were chosen because of their widespread utility in scientific and engineering applications. Actually, the trigonometric functions and their inverses are each represented by a pair of overloaded functions, with different numbers of parameters; the same is true of the LOG function. With overloadings included, a total of twenty-eight functions and one operator are defined in the elementary functions package.

*Why were the names X and Y chosen for the formal parameters of the "**" operator?*

Whenever one overloads an operator, one is strongly motivated to retain the existing names of its formal parameters. Had this practice been followed with respect to the overloading of "**" contained in GENERIC_ELEMENTARY_FUNCTIONS, the names LEFT and RIGHT would have been retained for its formal parameters. However, bowing to common practice, as well as to the precedent set by other standards (such as the IEEE standards for floating-point arithmetic), the committee had already chosen the name X for the formal parameter of the one-place functions, and the names X and Y for those of the two-place versions of ARCTAN and ARCCOT. For uniformity within this package, the names X and Y were therefore also used for the formal parameters of the "**" operator. The decision to do so was also influenced by the desire to use short names in the accuracy requirements, which in some cases are expressed in terms of the values of the formal parameters.

The overloading of "**" contained in GENERIC_ELEMENTARY_FUNCTIONS differs in another respect, as well, from the predefined "**" operator (see the question on the behavior of 0.0**0.0, below).

Are angles measured in radians, degrees, or what?

Users have a choice of units in which angles are measured, and the overloadings in the trigonometric family and their inverses play a role in the exercise of that choice.

Most often, the desired angular measure is radians; consequently, it is the easiest to specify. Thus, `SIN(X)`, for example, yields the sine of `X`, where `X` is understood to be measured in radians, and similarly `ARCSIN(X)` yields the angle (in radians) whose sine is `X`. To specify some other angular measure, one would supply a value for `CYCLE` (which is the second parameter, except in the case of `ARCTAN` or `ARCCOT`, where it is the third). For example, the sine of `X`, where now `X` is understood to be measured in degrees, would be written as `SIN(X, 360.0)`; by the same token, the angle (in degrees) whose sine is `X` is `ARCSIN(X, 360.0)`. Other angular measures can be accommodated by using the appropriate value for `CYCLE`—e.g., `6400.0` for mils, `400.0` for grads. From these examples it should be clear that the numerical value of `CYCLE` has the following interpretation: an angle `X` numerically equal to the value of `CYCLE` represents one complete cycle of revolution (i.e., one period of the function).[3] It should also be clear that when the `CYCLE` parameter is omitted, as in `SIN(X)`, the effect is as if a `CYCLE` of 2π had been specified.

A similar choice exists with respect to the base of the `LOG` function. `LOG(X)` means the natural or Napierian logarithm (i.e., base e); for other bases, such as `2.0` or `10.0`, which are perhaps the most common after e, one writes `LOG(X, 2.0)`, `LOG(X, 10.0)`, etc. The optional parameter is called `BASE`, and when it is omitted the effect is as if a `BASE` of e had been specified. There is no `BASE` parameter for the `EXP` function, which is the inverse of `LOG`, since that functionality can be obtained with the exponentiation operator: the number whose logarithm to the base `B` is `X` can be computed as `B**X`.

Why is the optionality of the CYCLE and BASE parameters handled by subprogram overloading instead of simply using default values for them?

The preceding discussion suggests that the optionality of the `CYCLE` and `BASE` parameters could have been handled very simply by defining appropriate default values for these parameters, and yet subprogram overloading was used instead, which unfortunately adds nine subprogram declarations to the specification of `GENERIC_ELEMENTARY_FUNCTIONS`. What is so disadvantageous about the default-value method to warrant this increase in the size of the specification of `GENERIC_ELEMENTARY_FUNCTIONS`?

The answer is rather subtle, and it required much debate during the development of this proposed standard. The problem is essentially that 2π and e, being irrational, are not representable exactly in any implementation, so the best that could possibly be done would be to use a default value that is a close approximation to 2π or e (implicit conversion of 2π or e, expressed as a numeric literal with an arbitrarily large number of trailing digits, to the type `FLOAT_TYPE` is only required to yield a value in the same safe interval as the literal). The use of such an approximation to 2π in the computation of the trigonometric functions, as if it were the true period, would produce results with unacceptable accuracy. The error is not like a simple roundoff error but has the nature of a cumulative phase shift that increases as the number of periods, or cycles, represented by `X` increases. For sufficiently large `X`, there will be no correct digits whatsoever in the result.

Better results can be obtained, and this problem ameliorated somewhat, if the implementation uses an internal representation of 2π that has more precision than `FLOAT_TYPE` has. The necessary additional precision may be obtained from hardware, if it is available; if not, there are techniques for "simulating" extra precision (see, e.g., [25]). But using extra precision only pushes the "phase shift" problem, wherein all accuracy is lost, farther away—i.e., to larger values of `X`; it does not eliminate it entirely. Therefore, the amount of extra precision required is related to the range of values of `X` for which some stated accuracy is to be achieved. Since the required accuracy and the minimum domain over which it must be achieved are both spelled out in the proposed standard, implementors have the information they need to satisfy the standard's requirements. (Outside the range of `X` for which the trigonometric functions with natural cycle must meet the accuracy requirements, degraded accuracy—but not other behavior, such as the raising of an exception—is allowed. An implementation must document the accuracy it achieves for extreme values of `X`, along with the threshold where degradation below the required accuracy commences.)

[3]The usage notes in drafts of the proposed standard preceding Draft 1.2 showed examples involving a `CYCLE` of `1.0`, allegedly corresponding to angular measure in bams ("binary angular measure"). It was tardily discovered [11] that this obscure invention is primarily useful in *fixed-point* implementations, where the primary range of the angle is taken to be $[-1, 1)$ bams, corresponding to $[-\pi, \pi)$ radians, and scaled to make use of all the bits in an integer word—for example, scaled to $[-32768, 32767]$. Thus, a full cycle in bams is `2.0`, not `1.0`. Since a `CYCLE` of `2.0` does not begin to suggest the real utility of bams, the examples involving bams were omitted in Draft 1.2 instead of being corrected.

It turns out, in fact, that if the default-value method had been employed for CYCLE in this standard, and an implementation used that value as if it were the true period, then the range of values of X for which the stated accuracy could be achieved would not even extend as far as one complete period away from the origin. This also means that an implementation that calculates SIN(X) by calling SIN(X, P) for some value of P meant to approximate 2π, including a literal with an arbitrarily large number of digits, will fail to meet the specifications.

All of the other standard periods are given by CYCLE values that are representable on all machines. With the aid of an appropriate "exact-remainder" algorithm, implementations of the explicit-CYCLE forms of the functions will have no difficulty reducing arbitrarily large values of X to the primary interval near the origin without error. (In fact, the exact-remainder function is included in a generic package of floating-point manipulation functions called GENERIC_PRIMITIVE_FUNCTIONS also being proposed for standardization; see [18].) That is why the explicit-CYCLE forms of the trigonometric functions have no restrictions on the range of values of X for which the stated accuracy requirements must hold.

It might have been reasonable to employ the default-value method for CYCLE and expect implementations to meet the accuracy requirements over the given range of values for X by recognizing when CYCLE has the default value but not using it (in that case) as if it were the true period—i.e., by using the default value merely as a *signal* that a different argument-reduction technique should be used. This was actually considered but ultimately abandoned because of the possibility of non-monotonic behavior as CYCLE sweeps through the default value, and because of potential surprises that could occur should users supply an explicit CYCLE that they believe to be a close approximation to 2π but that is not identical to the default value. Making CYCLE an enumeration type was also considered, but that, too, was abandoned, largely because it would have limited the available periods to a fixed set; furthermore, obtaining a numerical value of the period denoted by the enumeration value represents an unnecessary overhead.

Various other solutions of "the CYCLE problem" were also investigated, such as packaging the trigonometric functions and their inverses in an inner generic package having CYCLE as a generic formal parameter (e.g., a generic formal object of mode "in"). While these offered some elegant advantages, they also had unacceptable disadvantages.

The inverses of the trigonometric functions do not exhibit the "phase shift" phenomenon with respect to errors in their periods. For them, it is sufficient to compute the result as (for example) a fraction of a period and then scale that by multiplying by CYCLE. The default-value method for handling the optionality of CYCLE would have posed no problems, but subprogram overloading was preferred simply for reasons of uniformity. The same reasoning applies to BASE, in the case of LOG.

What purposes do the accuracy requirements serve, and how were they determined?

One of the significant advances represented by this proposal is its inclusion of accuracy requirements for implementations of the elementary functions. Not usually considered in formal specifications of mathematical software, accuracy requirements are made possible largely by the model (adapted from Brown [3]) of real arithmetic incorporated into Ada, and the form they take is influenced by the availability of attributes that characterize properties of Ada implementations relative to that model. Because the accuracy requirements will have an effect on what implementors can do, they will translate into some assurance of quality for the user; in addition, they will permit users to carry out error analyses of programs containing references to the elementary functions.

Two kinds of accuracy requirements—maximum-relative-error bounds and "prescribed results"—are included. All of the functions have maximum-relative-error bounds that limit the relative error in the computed result, over the whole range of valid arguments (or, in some cases, over a stated portion of the range). In addition, the required results at certain key argument values are prescribed more precisely for some of the functions.

The maximum-relative-error bounds were determined by numerical analysts having broad knowledge of algorithms and implementation techniques for the elementary functions. They are, of course, tailored to the specific properties of each function. They are considered to be realistic and to give implementors some leeway for creativity and individualism in regard to the trade-off between accuracy and efficiency. While they do rule out naïve implementations, they have proven to be conservative in the sense that it is not especially difficult for a knowledgeable implementor to produce implementations exceeding the accuracy requirements.

The maximum-relative-error bounds are based on the implemented precision of the generic actual parameter associated with FLOAT_TYPE rather than on its declared precision. This is reflected in the use of FLOAT_TYPE'BASE'EPSILON, rather than FLOAT_TYPE'EPSILON, in the formulas for maximum relative error. Effectively, those formulas constrain the computed result to lie within an appropriate number of safe intervals of the mathematical result, just as Ada does for the predefined arithmetic operators, which is what motivated the use of the BASE attribute in these formulas. In practice, it means that the approximation technique employed by the implementation must be appropriate for the precision of the base type of FLOAT_TYPE rather than just that of FLOAT_TYPE itself—i.e., it must be capable of exploiting all the precision inherent in the underlying base type.

Error analyses of programs containing the elementary functions, using the maximum-relative-error bounds as given in the specifications of GENERIC_ELEMENTARY_FUNCTIONS, are qualitatively portable in the sense that their form does not change from one Ada implementation to another. They are not quantitatively portable, however, since the numerical size of the maximum-relative-error bounds depends on the Ada implementation's mapping between user-declared floating-point subtypes and the predefined floating-point types. Nevertheless, a quantitatively portable error analysis can also be carried out merely by substituting FLOAT_TYPE'EPSILON for FLOAT_TYPE'BASE'EPSILON wherever it arises in the analysis. The choice is equivalent to carrying out the error analysis either at the level of safe numbers or at the level of model numbers; both are qualitatively portable, but only the latter is quantitatively portable. Ada gives analysts that choice when they can "see" all the way down to the level of the basic operations and predefined operators in their Ada programs; the proposed standard for the elementary functions preserves that choice—without requiring one to look inside implementations of the elementary functions—by constraining and describing their behavior at the level of safe numbers (from which their behavior at the level of model numbers can be trivially inferred).

A considerable amount of debate was necessary to reach consensus on this form of the maximum-relative-error bounds. Some of the contributors to this proposal felt, and still feel, that an implementation should be permitted to use coarser and coarser approximation methods as the precision of FLOAT_TYPE decreases (e.g., in different instantiations), even when the precision of its base type remains the same. For example, a graphics application might well not need 6-digit accuracy in the elementary functions when the user's generic actual subtype is declared as "digits 2", and the user might not be willing to pay for the unneeded accuracy in the form of additional iterations through some loop, or additional terms in an approximating polynomial, inside the body of an elementary function. A majority of the contributors felt that it was better to require software to get the most out of the hardware it is given to work with, at least for standard-conforming implementations, reasoning that special requirements can always be met by additional implementations not conforming to the standard.

There was also a question as to whether the use of the BASE attribute in the accuracy requirements adequately reflects the *implemented* precision of the user's generic actual subtype (for example, in the case of a reduced-accuracy subtype, perhaps combined with the influence of representation clauses). Ada Commentary AI-00407 [14] implies that the implemented precision of a reduced-accuracy subtype, as it affects the storage of variables of the subtype as well as parameter associations and function returns involving the subtype, may be less than the precision of the subtype's base type. Because that decision has profoundly undesirable consequences (including the obfuscation of the concept of "representation of a type"; the loss of the ability to specify and analyze the behavior of composite operations, represented by functions, using the same abstractions—including safe intervals—as are applicable to the basic and predefined operations; and the rendering of certain classes of attributes nearly useless), and because it appears to conflict with other requirements or implications of the language [27], some observers feel that it is ill advised. Accordingly, Ada Commentary AI-00571, which calls for the reevaluation of AI-00407, was submitted in July of 1988. WG9 returned AI-00407 to the Ada Rapporteur Group for reconsideration in June of 1989, and the ARG completed its reconsideration four months later by approving AI-00571 (thereby rescinding AI-00407). Thus, it is now known that the accuracy laboriously achieved in the body of an elementary function will not be thrown away at the return, even when the generic actual subtype is a reduced-accuracy subtype.

"Prescribed results" are used in some cases to constrain the computed result even more than the maximum-relative-error bounds constrain it. For example, EXP(0.0) is prescribed to yield exactly 1.0. The prescribed results reflect behavior that is both highly desirable from a numerical point of view and easy to achieve. In most cases, sensible algorithms will achieve the required behavior without extra effort; when necessary, it can always be achieved with a test for the special argument values.

Some of the prescribed results appear to require the function to deliver a value that cannot be computed exactly; for example, one of the prescribed results reads "ARCSIN(1.0) = $\pi/2$". What does this mean? The proposed

standard says that a prescribed result that is a safe number must be delivered exactly; in the case of one that is not, such as this one, the implementation may deliver any value in the surrounding safe interval. The required behavior can be achieved without difficulty, even in portable implementations.

Consideration was given to allowing the trigonometric functions with natural cycle to raise ARGUMENT_ERROR for sufficiently large X, where it is impractical to avoid accuracy degradation. However, this would have weakened the significance of ARGUMENT_ERROR as an implementation-independent indicator of mathematical domain violation. Raising a different exception was also considered, but in the end the committee thought that most users would prefer continuation with a result that falls short of the accuracy requirements by a known amount to no result at all.

What is the role of the range definitions?

Range definitions (or restrictions) are included with some of the functions for several reasons. In the case of functions that are mathematically multivalued, they serve to define the principal range for the implementation, enabling it to be single-valued without ambiguity. In other cases, they impose highly desirable and easily achieved numerical constraints on the results—constraints that do not automatically follow from the maximum-relative-error requirements. In this latter context, they behave like additional prescribed results (in the form of an inequality, rather than an equality). And like prescribed results, range limits are sometimes given by values that cannot be computed exactly. In analogy to the meaning of prescribed results, the proposed standard defines the meaning of range limits like this: When a range limit is a safe number, the implementation must not exceed it; when it is not, the implementation may exceed it, but it may not exceed the next safe number beyond the range limit in the direction away from the interior of the range. The required behavior can be achieved without difficulty, even in portable implementations. (For more on range definitions, see the question on underflow, below.)

How are exceptional conditions treated?

Two types of exceptional condition are explicitly recognized by the proposed standard; in each case, the defined action is to raise an exception. Equally important, an implementation is prohibited from raising spurious exceptions if it is to conform to the standard.

The first type of exceptional condition under which an implementation is allowed to raise an exception instead of delivering a result occurs when the arguments of one of the elementary functions are such that its mathematical result is not defined—in other words, when its arguments are invalid. A familiar example, given that arguments and results in this package are restricted to the real domain (as opposed to the complex domain), is an attempt to compute the square root of a negative number. The validity or invalidity of arguments is completely defined by the "domain definitions" included with the description of each function in the proposed standard.

When faced with invalid arguments, an implementation is not merely allowed to raise an exception; it is required to do so. The standard prescribes the raising of the ARGUMENT_ERROR exception, which it defines and reserves for this situation.

The validity of given arguments is never influenced by hardware properties; if ARGUMENT_ERROR is raised by a function for certain arguments in one implementation, it will be raised for those arguments in any implementation. Argument validity can reliably be established by inspection of the arguments, that is, by subjecting them to appropriate tests. While it will usually be most convenient for an implementation to check for argument validity before attempting to compute a result, other strategies may be possible and appropriate in some cases.

The second type of exceptional condition under which an implementation is allowed to raise an exception instead of delivering a result occurs when the mathematical result is well defined for the given arguments but something else (from among a limited list of things) unavoidably stands in the way of actually delivering that result. There are in fact three misfortunes that can befall an implementation, interfering with its ability to deliver a numerical result that is close enough to the mathematical result to satisfy the accuracy requirements:

- It may happen that the given arguments fail to satisfy range constraints inherent in the user's generic actual subtype, or that the function's computed result fails to satisfy those constraints.
- It may happen that the function that is invoked is unable to obtain the storage it needs to perform the requested computation.

- It may happen that the computed result is so large, in magnitude, that it exceeds the hardware's representational capabilities—i.e., it overflows.

The first of these misfortunes can occur during any parameter association or on any function return; it is not peculiar to the functions in this package. It is a fact of life of Ada, calling for the raising of the CONSTRAINT_ERROR exception at the place of the call when it occurs during a parameter association, or at the place of the return statement when it occurs during a function return. In the former case the function is never entered, so clearly a numerical result cannot be delivered. In the latter case the function has computed an appropriate numerical result and has attempted to deliver it but has failed, because of the range constraints that the user has imposed on the arguments and results of all the elementary functions. In marginal cases, other (slightly different) results might have been produced that do satisfy the range constraints while still satisfying the accuracy requirements, but it is not highly likely. So, for all practical purposes, it may be assumed that it is just not possible to deliver a satisfactory numerical result. The proposed standard for GENERIC_ELEMENTARY_FUNCTIONS, by allowing CONSTRAINT_ERROR to be raised naturally when this misfortune occurs, does not impose any special design requirements on implementations. Indeed, in the case of parameter associations, there is nothing else that it could do.

The second of the three misfortunes can occur during any subprogram invocation, or during the elaboration of a subprogram's declarative part just after its invocation; it, too, is not peculiar to the functions in this package. Thus it, too, is a fact of life of Ada, calling for the raising of the STORAGE_ERROR exception at the place of the call (for all practical purposes) when it occurs for either of these reasons. Clearly, it is just not possible to deliver a numerical result, since the function either has never been entered or has not finished elaborating its declarative part. Since storage requirements for elementary function implementations (of scalar arguments) are modest, it is highly likely that the application was running near the limit of available storage at the point where the elementary function was invoked, and that it is merely an accident that the raising of STORAGE_ERROR occurred there instead of somewhere else. The proposed standard for GENERIC_ELEMENTARY_FUNCTIONS, by allowing STORAGE_ERROR to be raised naturally when this misfortune occurs, does not impose any special design requirements on implementations. Indeed, since a handler for STORAGE_ERROR established by the function will not be entered in the cases described above, there is nothing else that it could do. (STORAGE_ERROR can also be raised by evaluation of an allocator in the sequence of statements of the function's body. Although in that case the propagation of the exception to the caller is *not* inevitable, it seemed hardly practical or appropriate for the proposed standard to distinguish that case from the earlier ones.)

The last of the three misfortunes, a computed result whose magnitude is so big that it cannot be represented in the hardware, is commonly called overflow. Since the overflow threshold is a hardware property, exactly which results are "too big" cannot be predicted with certainty without reference to the hardware. The model of real arithmetic in Ada allows one to say with certainty, however, that a computed result whose absolute value is less than or equal to FLOAT_TYPE'SAFE_LARGE is always capable of being represented. Thus it is reasonable to insist that, whenever all possible results permitted by the accuracy requirements are less than or equal to FLOAT_TYPE'SAFE_LARGE in absolute value, the implementation must deliver one of them (if it does not suffer one of the earlier misfortunes). On the other hand, if any result permitted by the accuracy requirements would exceed FLOAT_TYPE'SAFE_LARGE in absolute value, then it is possible that that result is the one that the implementation might try to compute, and that it would also exceed the hardware's overflow threshold. Thus, whenever any result permitted by the accuracy requirements exceeds FLOAT_TYPE'SAFE_LARGE in absolute value, an implementation is permitted to signal overflow instead of delivering a result. That does not mean that it *must*, of course; the actual result that it computes could be some other permitted result that does not exceed FLOAT_TYPE'SAFE_LARGE in absolute value—and therefore does not exceed the hardware's overflow threshold—or it could be one that does exceed FLOAT_TYPE'SAFE_LARGE but still does not exceed the hardware's overflow threshold. By the same token, the fact that some of the permitted results do not exceed FLOAT_TYPE'SAFE_LARGE in absolute value does not oblige the implementation to deliver a result in the case that others do exceed it. This is the rationale for the variety of behaviors that an implementation is allowed to exhibit in the vicinity of the overflow threshold.

If overflow needs to be signaled, the proposed standard calls for that to be done in the way that Ada mandates for its predefined operators—i.e., by raising NUMERIC_ERROR (which is changed to CONSTRAINT_ERROR by AI-00387).

Implementors have a choice of ways to deal with possible overflows in the final result. On the one hand, implementors can use a predictive technique—that is, examine the arguments before using them to compute the result and raise the appropriate exception (by a "raise" statement) if one of the permitted results (i.e., any value differing from the mathematical result by no more than the maximum relative error) would exceed FLOAT_TYPE'SAFE_LARGE

in absolute value. Or, implementors can omit the argument prescreening and just go ahead and compute, relying on the hardware to detect and signal in the natural way an overflow in the final result—or an overflow (or other exceptional condition) that presages overflow in the final result, even though it may occur well before the final step in the process of obtaining that result. The latter technique entails a more sophisticated analysis on the part of the implementor, who must be certain that extreme arguments do not violate some implicit assumption of the algorithm, causing it to misbehave and leading to an unacceptable numerical result rather than to overflow, but that extra effort is usually rewarded by a marginally more efficient and useful product.

Recall that, whenever all possible results permitted by the accuracy requirements are less than or equal to `FLOAT_TYPE'SAFE_LARGE` in absolute value, the implementation must deliver one of them (if it does not suffer one of the earlier misfortunes). Through that provision the proposed standard imposes on implementors the responsibility of ensuring either that their chosen algorithm does not inadvertently overflow (or suffer some other numerical exception) in the calculation of an intermediate result or that, if it does, it handles the exception locally and goes on to compute and deliver an acceptable result. To conform to this standard, an implementation cannot be so naïve that it raises spurious exceptions.

The three "misfortunes" described above share two characteristics: (1) they are not peculiar to the elementary functions and can indeed occur in many places in arbitrary Ada programs, and (2) Ada provides predefined exceptions to be used to signal their occurrence. It seemed inappropriate for this standard to distinguish their occurrence in the context of the elementary functions by prescribing the raising of a new exception (a possibility that was considered); it would not have been possible to do so uniformly and consistently, in any case.

In fact, consideration was even given to carrying the preceding decision to its extreme, viz., to dispensing with the `ARGUMENT_ERROR` exception and instead prescribing the raising of `CONSTRAINT_ERROR` for invalid arguments, on the grounds that an invalid argument represents the violation of a constraint in the broad sense of the term. While that treatment of invalid arguments would parallel the handling of division by zero (which originally raised `NUMERIC_ERROR`, but under the influence of AI-00387 now raises `CONSTRAINT_ERROR`), it was felt that prescribing the use of a new exception to report exceptional conditions *uniquely related to the functionality of this package*, following the precedent set by the predefined I/O packages, was more useful to the application programmer.

The name `ARGUMENT_ERROR` exported by instantiations of `GENERIC_ELEMENTARY_FUNCTIONS` is a renaming of the exception of the same name declared in the package `ELEMENTARY_FUNCTIONS_EXCEPTIONS`, whose specification is also contained in this proposed standard. Therefore, multiple instantiations, should they be needed in the same application, do not give rise to multiple `ARGUMENT_ERROR` exceptions. Thus it is not necessary to establish a handler for multiple instances of this exception; one handler suffices for all the instantiations. Unfortunately, this convenience is partially offset by the fact that the name `ARGUMENT_ERROR` will not be directly visible in the case of multiple instantiations, and an expanded name will have to be used. The alternative of not exporting the name `ARGUMENT_ERROR` from instantiations, and requiring the user to "`with`" and "`use`" `ELEMENTARY_FUNCTIONS_EXCEPTIONS` to gain direct visibility of the name `ARGUMENT_ERROR`, was considered, but it was rejected on the grounds that the present proposal is more convenient for the user when there is only one instantiation, as is expected to be the common case. The precedent set by the exporting of renamed exceptions by some of the predefined I/O packages, which unfortunately suffer from the same name-visibility problem, also figured in this decision.

If overflow is signaled by an exception, why isn't underflow so signaled?

The proposed standard does not call for the raising of an exception upon underflow primarily because of the precedent set by Ada, which does not have a predefined exception for underflow. In fact, underflow is not considered an exceptional condition by Ada, by virtue of the fact that the interval between zero and `T'SAFE_SMALL` (for any real subtype `T`) is a normal safe interval. This treatment by Ada is consonant both with the returning of "denormalized numbers" by hardware obeying the IEEE standards for floating-point arithmetic and with the classical recovery from hardware underflow, namely, flushing to zero.

The possibility of underflow and, in particular, flushing to zero, introduces the chance that the computed result of some of the elementary functions might be `0.0` even though `0.0` is not in their mathematical range for any finite argument. For example, `EXP` of an extremely negative argument might underflow to `0.0`, even though e^{X} cannot mathematically be exactly zero for any finite `X`. Similarly, `X**Y` can underflow to `0.0` for positive `X` and extremely negative `Y`, even though mathematically X^{Y} cannot be exactly zero for any finite `Y` (when `X` is nonzero). Another

disadvantage of flushing to zero is that it fails to satisfy small-relative-error requirements. For these reasons, an earlier version of the proposed standard actually called for the raising of an exception, rather than returning zero, in underflow situations—for those functions whose mathematical range does not include zero. That was abandoned, however, because it would have been too radical a departure from common practice, not to mention inconsistent with Ada's handling of the predefined operators.

The proposed standard substitutes a small-absolute-error requirement for the small-relative-error requirement in underflow situations (where the latter is unsatisfiable). Furthermore, the range definitions in the proposed standard include asymptotic limits. Their inclusion not only legalizes the returning of `0.0` in the cases described above, but also legalizes the returning of $\pm$`1.0` in the case of sufficiently extreme arguments to `TANH` or `COTH`; this is desirable because, even though there is no finite `X` for which the hyperbolic tangent or cotangent of `X` is exactly `+1.0` or `-1.0`, these values may be closer to the mathematical result than any other representable number.

Why does 0.0**0.0 raise ARGUMENT_ERROR?

The point `X` = `Y` = `0.0` is excluded from the domain of `X**Y` because it is not possible to assign a unique value to the result, or to pick a conventional value that is suitable for all applications. Essentially, this acknowledges the fact that the limiting value of X^Y as both `X` and `Y` approach `0.0` depends on exactly *how* `X` and `Y` approach `0.0`.

The committee considered defining the result of `0.0**0.0` in this standard to be `1.0`, as advocated by some numerical analysts [19]; this would have had the virtue of agreeing with the predefined `"**"` operator (whose right operand has type `INTEGER`). However, the committee defended its decision on the grounds that exponentiation by an integer and exponentiation by a real number are two different functions, as reflected in their conventional definitions (in terms of repeated multiplication in the former case, and exponentials and logarithms in the latter).

The consequences of the committee's decision (regardless of which way it went, ultimately) were softened by the realization that individual programmers could easily enough obtain any other behavior required by their application by putting an appropriate shell around `"**"`. Thus, in the final analysis, the committee focused on what seemed most desirable for the default behavior of `0.0**0.0`, and it concluded that the most conservative and safest choice was the most desirable. The decision to raise `ARGUMENT_ERROR` was deemed to provide the safest default behavior because, in those rare applications in which `0.0**0.0` actually could arise, the raising of `ARGUMENT_ERROR` forces the programmer to *think* about the mathematics of the application.

The alternative philosophy, that of providing a numerical result (like `1.0`) by convention, on the grounds that it is desirable for at least some applications, is appropriate for a language lacking exceptions and exception handlers, especially when it is backed up by something like a "sticky" flag to allow detection of the case when it occurs but is *not* desired (without imposing an undue cost on the user when it *is* desired); it is much more in the conservative style and culture of Ada, however, not to impose particular programming disciplines on users and to provide only minimal and safe capabilities out of which the user can fashion whatever discipline is appropriate.

How are portable implementations of GENERIC_ELEMENTARY_FUNCTIONS accommodated?

Two kinds of implementations of `GENERIC_ELEMENTARY_FUNCTIONS` are envisioned. On the one hand, vendors of Ada compilers for specific hardware might have an interest in producing tailored implementations of `GENERIC_ELEMENTARY_FUNCTIONS` for that hardware; for such implementations, portability is not a concern, and major portions of those implementations might not even be written in Ada. On the other hand, independent software producers could have an interest in developing a single implementation of `GENERIC_ELEMENTARY_FUNCTIONS` that is portable to a wide variety of different machines and Ada systems. While both kinds are allowed by the specifications, portable implementations—understandably the more challenging of the two—are affected by certain special considerations that do not apply to tailored ones (but they also benefit from some mild concessions, like the ones inherent in the treatment of prescribed results or range limits that are not safe numbers).

It is highly unlikely that an implementation of `GENERIC_ELEMENTARY_FUNCTIONS` can be rendered in Ada so as to be portable without qualification to all implementations of Ada. Typically, implementations intended to be portable are not absolutely portable, but are portable only with certain qualifications stemming from assumptions

built into the code. Some examples of assumptions that simplify and circumscribe the design of an implementation of the elementary functions are the following:

- It might be assumed (i.e., required) that `FLOAT_TYPE'MACHINE_RADIX` is 2 or 16, and not some other number.
- It might be assumed that multiplication or division by a power of the machine radix is exact.
- It might be assumed that `SYSTEM.MAX_DIGITS` falls within some particular range of values.

Assumptions are always avoidable, or at least capable of being weakened, but the amount of effort required to do so may be more than the implementor can justify.

As long as "portable" implementations embody assumptions, there is a risk that they will fail in unpredictable ways when used in an unintended environment (where the assumptions may not hold). The proposed standard addresses this risk in three ways. First of all, it requires that the assumptions be clearly documented, enabling potential users of an implementation to evaluate its suitability in their environment. Secondly, it does not define the behavior of an implementation whose assumptions are violated, thereby giving implementors total freedom to opt either for speed (by ignoring the possibility of such violations) or for robustness (by detecting them and responding—predictably—in ways of their choice). (For example, an implementation might raise `PROGRAM_ERROR`, or some other exception, upon detecting a violation of its assumptions.) Finally, the proposed standard adopts the convention that implementations whose assumptions are violated in some environment are not in conformance with the standard in that environment. This last point is intended to discourage highly restricted (partial) implementations from absolving themselves in the guise of portability. It is not enough to be portable; an implementation must also be useful, and usefulness is measured by the number of environments in which it is conforming.

It is sufficient for a non-portable implementation (one intended for a single environment) to document its assumptions as "intended for use only in the [*name of the environment*] environment."

What role do "signed zeros" and infinities play in the elementary functions?

One of the features of the IEEE standards for floating-point arithmetic is that the value zero is not automatically canonicalized with the conventional plus sign; it is thus capable of acquiring either algebraic sign. The prescribed sign of a zero result depends on context—the operation, the operands, the "rounding mode," etc.—and the rules are spelled out in the IEEE floating-point standards for the operations covered by them. In particular, a zero arising from underflow when traps are disabled preserves the sign of the result that underflowed; this rule and others reveal that signed zeros behave in many contexts like correctly signed infinitesimal quantities.

The IEEE floating-point standards address both the determination of the sign of a zero result when it is generated and the use of the sign of a zero operand when it is consumed. Any software standard that wishes to build upon and be consistent with the IEEE floating-point standards should address the same two points in relation to the extended "operations" that it provides, and the elementary functions standard is no exception. At the same time, support for signed zeros must not be a mandatory feature, since IEEE arithmetic is not universally available.

The handling of signed zeros in the proposed elementary functions standard came under consideration relatively late (after Draft 1.1, which means after approval by WG9); it was prompted by ongoing discussions concerning the development of an IEEE binding for Ada [5]. One of the topics of discussion has been *the division of responsibility:* how much of the treatment of signed zeros should be written into the elementary functions standard (and related standards), and how much should be written into the IEEE binding? Placing the treatment of signed zeros close to the functions affected has the advantage that it can be easily tailored to the requirements of the individual functions and the disadvantage that consistency of treatment among a family of related standards is difficult to ensure. Placing their treatment only in the IEEE binding has the complementary advantages and disadvantages, together with a few additional advantages: the optional nature of the handling of signed zeros in secondary standards like the elementary functions standard can be tied more readily to the availability of IEEE arithmetic, and the secondary standards themselves can remain relatively uncluttered. Since the issue is ultimately more a matter of implementation guidelines than of definition, it is reasonable to expect the WG9 Uniformity Rapporteur Group (URG) to have the final word on it.

The compromise approach followed in the elementary functions standard is as low-key as it could reasonably be made. Since a precise definition of what comprises the ability to represent and discriminate between positively and negatively signed zeros was judged to be beyond the scope of the elementary functions standard (for example,

it is not clear that such an ability should be tied too closely to the availability of full IEEE arithmetic), support for signed zeros by implementations of the standard was left strictly optional, even when the underlying capability can reasonably be said to be available. Furthermore, it has been left to other documents (such as the IEEE binding or a report of the URG) to specify in the future the sign of each possible zero result and the conditions upon which the sign depends. Such information would constitute a *refinement* of the zero results prescribed or implied by the elementary functions standard and would not in any way conflict with them.

The one place, however, where signed zeros do intrude into the elementary functions standard is in certain prescribed results from ARCTAN and ARCCOT. As an example, consider ARCTAN(Y, X) when Y is zero and X is negative. In this case, the point (X, Y) is on the axis separating quadrants II and III, so the corresponding angle is either π or $-\pi$. To make the function single-valued, drafts of the standard prior to Draft 1.2 prescribed this result to be (by convention) π. However, if one chooses to take the sign of Y into account and to interpret signed zeros as signed infinitesimals (on the assumption that they arise from underflow), then it is appropriate to prescribe the result to be π when the sign is positive and $-\pi$ when it is negative. Because that would be in conflict with the result prescribed in earlier drafts, the result prescribed in Draft 1.2 for ARCTAN(Y, X) when Y is zero and X is negative was loosened to $\pm\pi$. The choice between $-\pi$ and π depends on whether signed zeros are being supported and, if they are, on the sign of Y; in the latter case, the appropriate interpretation of the result is the limit of ARCTAN(ϵ, X), for negative X, as ϵ approaches zero, which depends on the sign of ϵ, or equivalently on the quadrant from which the axis separating quadrants II and III is approached. Of course, similar considerations apply to ARCCOT and to the versions of both that include a CYCLE parameter; the standard requires that they be handled consistently. This treatment of signed zeros recognizes the variety of opinions on their utility; thus, it covers not only the architectural variations in the ability to support them, but also the implementor's preferences when architectural limitations are not a factor.

Detecting the sign of zero is not accomplished easily in Ada, since in IEEE arithmetic the sign of zero has no bearing on the outcome of comparison operations. One technique, which hinges on a few reasonable assumptions, involves looking at the bit pattern of a value that compares equal to zero and asking whether it matches that of plus zero or that of minus zero. The bit patterns are obtained with the help of UNCHECKED_CONVERSION, while a negative zero is obtained, at least on IEEE hardware, by computing a negative result that underflows. (Whether the underlying implementation generates negative zeros in this context can be determined by forcing both positive and negative underflows and comparing the bit patterns of the two resulting zeros.) Devious tricks such as this can be avoided by relying instead on a specific function, COPY_SIGN, from the proposed GENERIC_PRIMITIVE_FUNCTIONS package [18]. COPY_SIGN(VALUE, SIGN) is defined to yield a result having the magnitude of VALUE and the sign of SIGN (even when SIGN is zero). COPY_SIGN may itself be implemented in terms of the technique described above, but since implementations of GENERIC_PRIMITIVE_FUNCTIONS will commonly be tailored to the underlying hardware, it is more likely to be implemented in assembler language or C so as to exploit knowledge of the hardware in the most efficient way. While COPY_SIGN can be used in GENERIC_ELEMENTARY_FUNCTIONS to determine whether a zero is a positive zero or a negative zero, in practice that question need not be asked directly. For example, the prescribed results for ARCTAN, including the one discussed above, might be produced by code like the following:

```
with GENERIC_PRIMITIVE_FUNCTIONS;
package body GENERIC_ELEMENTARY_FUNCTIONS is
   ...
   package PRIMITIVE_FUNCTIONS is new GENERIC_PRIMITIVE_FUNCTIONS (FLOAT_TYPE, INTEGER);
   use PRIMITIVE_FUNCTIONS;
   PI : constant := 3.1415926535...E0;
   ...
   function ARCTAN (Y : FLOAT_TYPE;
                    X : FLOAT_TYPE := 1.0) return FLOAT_TYPE is
   begin
      if Y = 0.0 then
         if X = 0.0 then
            raise ARGUMENT_ERROR;
         elsif X < 0.0 then
            return COPY_SIGN(PI, Y);  -- (*)
```

```
            else  -- X > 0.0
               return Y;                    -- (**)
            end if;
         elsif X = 0.0 then
            return COPY_SIGN(PI/2.0, Y);
         else  -- end of special cases
            ...
         end if;
      end ARCTAN;
      ...
   end GENERIC_ELEMENTARY_FUNCTIONS;
```

An implementor of GENERIC_ELEMENTARY_FUNCTIONS who wishes to avoid signed zeros entirely, even when they are available, can do so merely by substituting "`return PI;`" for the statement marked (*) and "`return 0.0;`" for that marked (**).[4]

Like ARCTAN and ARCCOT in GENERIC_ELEMENTARY_FUNCTIONS, COPY_SIGN in GENERIC_PRIMITIVE_FUNCTIONS has a behavior that depends in one case on the sign of a zero argument. For the sake of uniformity, implementations of the latter are also allowed to forego the recognition of signed zeros, in which case COPY_SIGN must return |VALUE| when SIGN is zero. Thus, unless one knows something about the behavior of the implementation of GENERIC_PRIMITIVE_FUNCTIONS at hand, its use in an implementation of GENERIC_ELEMENTARY_FUNCTIONS is no guarantee that the latter will observe signed zeros in environments having that capability. Implementations of both of the packages discussed here are required to document their behavior with respect to signed zeros. Thus, an implementation of GENERIC_ELEMENTARY_FUNCTIONS might document that it does not observe signed zeros—even in environments having the capability—or that it does observe signed zeros in environments having the capability or that it inherits its behavior from other facilities that it uses (e.g., from GENERIC_PRIMITIVE_FUNCTIONS). Other software built on top of GENERIC_ELEMENTARY_FUNCTIONS has the same choices.

Signed zeros are naturally related to another feature of IEEE arithmetic—namely, (signed) infinities. Infinities arise in IEEE arithmetic, when traps are disabled, as a result of overflow and upon division by zero. The proposed IEEE binding [5] introduces infinities into Ada by specifying that the MACHINE_OVERFLOWS attribute will be FALSE in implementations conforming to the binding, which allows them to deliver an infinity instead of raising an exception when overflow occurs (and also when division by zero occurs, though this might require a clarification). GENERIC_ELEMENTARY_FUNCTIONS is well poised to exploit infinities in cases of overflow, if and when they are introduced by an IEEE binding. For example, in the context of an IEEE binding, no changes are necessary in GENERIC_ELEMENTARY_FUNCTIONS to allow EXP to deliver an infinity when it receives a sufficiently large argument, instead of raising an exception to signal overflow. Similarly, TAN and COT can deliver an infinity, instead of raising an exception to signal overflow, for arguments near their poles. However, prior to Draft 1.2 of the proposed standard, the machine-representable poles of functions (i.e., the machine-representable arguments for which the corresponding mathematical functions are infinite) were technically and formally excluded from the domains of the functions, which meant that an implementation must raise ARGUMENT_ERROR at the poles. This requirement was viewed as interfering with the ability to deliver an infinity at a pole, in the context of an IEEE binding. Accordingly, in Draft 1.2 the poles of functions were formally included in their domain, so that ARGUMENT_ERROR would no longer be raised there, and it was specified that the exception used by Ada for signaling division by zero would be raised instead at poles. Thus, the net effect of this change was merely to substitute one exception for another. In addition to facilitating the exploitation of infinities by GENERIC_ELEMENTARY_FUNCTIONS, if and when they are introduced by an IEEE binding, the change results in uniform behavior of functions both *near* poles and *at* poles, since the exception Ada uses for signaling division by zero is the same as the one it uses for signaling overflow.[5] This uniformity of behavior—that is, the raising of the same exception both near and at poles, rather than different

[4]Incidentally, note that the zero result produced by the statement marked (**) when Y is zero and X is positive retains the sign of Y—or so one hopes. There is no question that, in the absence of signed zeros, an optimizing compiler could compile the statement marked (**) as if it read "`return 0.0;`". However, the question of whether such an optimization should be allowed in the presence of signed zeros has not yet been definitively answered. A safer approach might be to write "`return COPY_SIGN(0.0, Y);`" at (**).

[5]The exceptional condition at poles is described as an occurrence of division by zero, rather than of overflow, because experience has shown that natural algorithms actually do cause a division by zero at poles. Note that it is possible to use the facilities of the proposed IEEE binding to establish different handlers for division by zero and for overflow.

exceptions—is appealing because it emphasizes the similarities in these two cases, rather than their differences (which might in fact stem from nothing more than certain machine dependences).[6]

Why is a package of mathematical constants not included in this standard?

A package of useful mathematical constants containing values for π, e, and many other things is an obvious candidate for inclusion in a numerical standard. While it would do little to increase the portability of numerical software (after all, one can look up values for these constants easily enough and write them as literals in programs, preferably in declarations of named numbers), such a package would—by eliminating the likelihood of transcription errors—potentially have a beneficial effect on the reliability of applications. Indeed, during the development of the `GENERIC_ELEMENTARY_FUNCTIONS` standard, the committee had numerous requests to include such a package and numerous suggestions for its contents. There was, at one time, an even more important reason for including it: until the decision was made to accommodate arbitrary periods in the trigonometric functions by subprogram overloading instead of by optional parameters with default values, the specification for `GENERIC_ELEMENTARY_FUNCTIONS` itself had a need to incorporate values for 2π and e, and it would have been important to allow that specification and the user's application to obtain them from the same source.

The specification in its final form does not have a need for those values. Nevertheless, that is not the reason for the absence of a package of mathematical constants in this standard; it is merely an explanation for why the specification of `GENERIC_ELEMENTARY_FUNCTIONS` can get by without one. The committee, in fact, early invested considerable effort towards the inclusion of such a package. The problem that it faced was in determining where to draw the line on its contents. When it became clear that no consensus was likely to be reached in a reasonable amount of time, the committee decided to defer consideration of a package of mathematical constants until some future time rather than risk delaying the development of the elementary functions standard unnecessarily.

Conclusions; a look at the future

The proposed elementary functions standard satisfies one part of Work Item JTC1.22.10.02 ("Standardization of Ada Numeric Packages"), assigned to the WG9 Numerics Rapporteur Group. Though it will significantly improve the prospects of portability for engineering and scientific applications written in Ada, much more is needed; accordingly, other standards are being developed in response to the remaining parts of the work item. Reference has already been made to the need for, and development of, a standardized generic package of floating-point manipulation functions, and to a package of mathematical constants. In addition to those, the people who have worked together on the elementary functions standard are now working on proposals for complex arithmetic and complex elementary functions, for vector and matrix routines, and for random-number generators. These efforts, whose motivation comes partly from the consensus of support already exhibited for the proposed elementary functions standard, can be expected to reach fruition in the next few years.

The focus on Ada numerics associated with these ongoing standardization activities is also fostering several research endeavors. One line of research growing out of this effort involves the development of testing strategies and algorithms that can be used to validate the conformance of implementations of the proposed standard. Such methods are under development at several locations, including ANL [24], NAG, CWI [2], and Westinghouse [23]; the work at NAG [8] also encompasses methods for checking that the underlying floating-point arithmetic of the host system conforms to the Ada standard. Other research in progress is aimed at using automated program verification techniques to prove accuracy claims for Ada floating-point programs. These research activities, taken together with the standardization efforts, will enhance both the theoretical and the practical aspects of Ada numerics.

[6]The computation of an argument that, on one machine, yields a value near a pole of the function being called might, on another machine (having a different word length or different rounding behavior), yield a value that coincides with the pole.

References

[1] Ada 9X Project. Ada 9X Requirements. Office of the Under Secretary of Defense for Acquisition, Washington, December 1990.

[2] M. Bergman. Testing of Elementary Functions in Ada. CWI Report NM-R8909, Centrum voor Wiskunde en Informatica, Amsterdam, May 1989.

[3] W. S. Brown. A Simple but Realistic Model of Floating-Point Computation. *TOMS*, 7(4):445–480, December 1981.

[4] J.-M. Chebat. Personal communication, March 21, 1990.

[5] R. B. K. Dewar. Proposed Ada Interface for the IEEE Standard for Binary Floating-Point Arithmetic, August 1989. Version 4.

[6] K. W. Dritz. Rationale for the Proposed Standard for a Generic Package of Elementary Functions for Ada. ANL Report ANL-89/2, Argonne National Laboratory, Argonne, Illinois, March 1989.

[7] K. W. Dritz. Influence of Language Features on the Specification of a Standardized Generic Package of Elementary Functions for Ada. In G. Pollock, editor, *Proc. 1988 Sandia Workshop on Ada in Real-Time and Scientific Environments*. Sandia National Laboratory, to appear. Available from author as ANL Preprint MCS-P16-1188.

[8] J. J. Du Croz, M. Erl, P. P. Gardner, G. S. Hodgson, and M. W. Pont. Validation of Numerical Computations in Ada. NAG Technical Report TR2/89, Numerical Algorithms Group, Ltd., Oxford, May 1989.

[9] R. Firth. Preliminary Draft Specification of a Basic Mathematical Library for the High Order Programming Language Ada. Royal Military College of Science, Shrivenham, Swindon, Wiltshire, England, March 1982.

[10] B. Ford, J. Kok, and M. W. Rogers, editors. *Scientific Ada*. Cambridge University Press, Cambridge, 1986.

[11] T. J. Froggatt. Are Angles Measured in Degrees, Radians, or What? *Ada User*, 11(3):106, 1990. Letter to the editor.

[12] IEEE. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std. 754-1985, IEEE, New York, 1985.

[13] IEEE. IEEE Standard for Radix-Independent Floating-Point Arithmetic. ANSI/IEEE Std. 854-1987, IEEE, New York, 1987.

[14] ISO-IEC/JTC1/SC22/WG9 (Ada) Ada Rapporteur Group. Ada-Comment Database. Maintained by the Ada Joint Program Office on AJPO.SEI.CMU.EDU.

[15] ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group. Proposed Standard for a Generic Package of Elementary Functions for Ada, March 1989. Draft 1.0.

[16] ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group. Proposed Standard for a Generic Package of Elementary Functions for Ada, October 1989. Draft 1.1.

[17] ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group. Proposed Standard for a Generic Package of Elementary Functions for Ada, December 1990. Draft 1.2.

[18] ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group. Proposed Standard for a Generic Package of Primitive Functions for Ada, December 1990. Draft 1.0.

[19] W. Kahan. Why Must $0^0 = 1$? Preprint, University of California at Berkeley, August 1988.

[20] J. Kok. Proposal for Standard Mathematical Packages in Ada. CWI Report NM-R8718, Centrum voor Wiskunde en Informatica, Amsterdam, November 1987.

[21] J. Kok and G. T. Symm. A Proposal for Standard Basic Functions in Ada. *Ada Letters*, 4(3):44–52, November/December 1984.

[22] R. F. Mathis. Elementary Functions Packages for Ada. In *Proc. 1987 ACM SIGAda International Conference on the Ada Programming Language* (special issue of *Ada Letters*), pages 95–100, December 1987.

[23] J. Squire. Validation of Reusable Numerical Software in Ada. Independent Research and Development Report, Westinghouse Electronic Systems Group, Baltimore, 1989 (to appear).

[24] P. T. P. Tang. Accurate and Efficient Testing of the Exponential and Logarithm Functions in Ada. ANL Report ANL-88-24, Argonne National Laboratory, November 1988.

[25] P. T. P. Tang. Portable Implementation of a Generic Exponential Function in Ada. ANL Report ANL-88-3, Argonne National Laboratory, February 1988.

[26] P. T. P. Tang. Use of Language Features in the Implementation and Validation of a Standardized Generic Package of Elementary Functions in Ada. In G. Pollock, editor, *Proc. 1988 Sandia Workshop on Ada in Real-Time and Scientific Environments*. Sandia National Laboratory, to appear. Available from author as ANL Preprint MCS-P25-1188.

[27] U. S. Department of Defense, Ada Joint Program Office. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A. U. S. Government Printing Office, Washington, D. C., 1983. Also adopted by ISO as ISO/8652-1987, Programming Languages—Ada.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	Final: December 1994

4. TITLE AND SUBTITLE

PROPOSED STANDARD FOR A GENERIC PACKAGE OF ELEMENTARY FUNCTIONS FOR ADA
Draft 1.2 ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group

5. FUNDING NUMBERS

PE: 0604711N
PN: X1144-CC

6. AUTHOR(S)

G. Myers, Chair

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Ocean Systems Center
San Diego, California 92152–5000

8. PERFORMING ORGANIZATION REPORT NUMBER

TD 2180

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Space and Naval Warfare Systems Command
Code SPWR–06
Washington, DC 20363

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The proposed standard for the Generic Package of Elementary Functions (GPEF) represents the work of a large number of people in both the United States and Europe who have collaborated to develop specifications for packages of Ada mathematical functions. This development has been difficult and lengthy. The exceptional dedication and perseverance of these people have resulted in the completed specifications for two packages—GPEF, and the Generic Package of Primitive Functions for Ada (GPPF).

GPEF is the specification for certain elementary mathematical functions. They are square root, logarithm and exponential functions and the exponentiation operator; the trigonometric functions for sine, cosine, tangent and cotangent and their inverses; and the hyperbolic functions for sine, cosine, tangent, and cotangent together with their inverses.

14. SUBJECT TERMS

C^3I Architecture
computers
programming languages

15. NUMBER OF PAGES

66

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAME AS REPORT

21a. NAME OF RESPONSIBLE INDIVIDUAL	21b. TELEPHONE *(include Area Code)*	21c. OFFICE SYMBOL
G. Myers	(619) 553–4136	Code 4104

INITIAL DISTRIBUTION

Code 0012	Patent Counsel	(1)
Code 0274	Library	(2)
Code 0275	Archive/Stock	(6)
Code 413	G. B. Myers, Jr.	(25)

Defense Technical Information Center
Alexandria, VA 22304–6145 (4)

NCCOSC Washington Liaison Office
Washington, DC 20363–5100

Center for Naval Analyses
Alexandria, VA 22302–0268

Navy Acquisition, Research and Development
Information Center (NARDIC)
Arlington, VA 22244–5114

GIDEP Operations Center
Corona, CA 91718–8000